ECHELON®

2.0

Neuron® Assembly Language Reference

078-0399-01A

Echelon Corporation
www.echelon.com

Welcome

Echelon's Neuron® assembly language is the symbolic programming language for both Series 3100 and Series 5000 Neuron Chips and Smart Transceivers. You can write a Neuron assembly language function or program that interacts with a Neuron C application program to provide LONWORKS® networking for new or existing smart devices. The Neuron assembly language is not intended as a general programming language for LONWORKS devices, but should be used only to optimize new or existing Neuron C applications.

This document describes the Neuron assembly language. The Neuron assembly language can be used with any programmable Series 3100 device (Neuron 3120® Chip, FT 3120 Smart Transceiver, PL 3120 Smart Transceiver, Neuron 3150® Chip, FT 3150 Smart Transceiver, PL 3150 Smart Transceiver, and PL 3170™ Smart Transceiver) and with any programmable Series 5000 device (FT 5000 Smart Transceiver and Neuron 5000 Processor). Where applicable, this document identifies differences in the Neuron assembly language that are specific to a particular device series.

Audience

This document assumes that you have a good understanding of general assembly language programming concepts and techniques. It also assumes that you are familiar with the Neuron C programming language and LONWORKS device development. In addition, a general understanding of either the Series 3100 or Series 5000 Neuron architecture is required.

Related Documentation

The following manuals are available from the Echelon Web site (www.echelon.com) and provide additional information that can help you develop applications for Neuron Chip or Smart Transceiver devices:

- *Series 5000 Chip Data Book* (005-0199-01A). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the Neuron 5000 Processors and FT 5000 Smart Transceivers.
- *I/O Model Reference for Smart Transceivers and Neuron Chips* (078-0392-01A). This manual describes the I/O models that are available for Series 3100 and Series 5000 devices.
- *FT 3120 / FT 3150 Smart Transceiver Data Book* (005-0139-01D). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the FT 3120® and FT 3150® Smart Transceivers.
- *Introduction to the LONWORKS Platform* (078-0391-01A). This manual provides an introduction to the ISO/IEC 14908 (ANSI/CEA-709.1 and EN14908) Control Network Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

- *LONMARK® Application Layer Interoperability Guidelines.* This manual describes design guidelines for developing applications for open interoperable LONWORKS devices, and is available from the LONMARK Web site, www.lonmark.org.

- *LonMaker User's Guide* (078-0333-01A). This manual describes how to use the Turbo edition of the LonMaker® Integration Tool to design, commission, monitor and control, maintain, and manage a network.

- *NodeBuilder® FX User's Guide* (078-0405-01A). This manual describes how to develop a LONWORKS device using the NodeBuilder tool.

- *Mini FX User's Guide* (078-0398-01A). This manual describes how to use the Mini FX Evaluation Kit. You can use the Mini FX Evaluation Kit to develop a prototype or production control system that requires networking, or to evaluate the development of applications for such control networks using the LONWORKS platform.

- *Neuron C Programmer's Guide* (078-0002-01H). This manual describes how to write programs using the Neuron C Version 2.2 programming language.

- *Neuron C Reference Guide* (078-0140-01F). This manual provides reference information for writing programs using the Neuron C Version 2.2 programming language.

- *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book* (005-0193-01A). This manual provides detailed technical specifications on the electrical interfaces, mechanical interfaces, and operating environment characteristics for the PL 3120, PL 3150, and PL 3170™ Smart Transceivers.

All of the Echelon documentation is available in Adobe® PDF format. To view the PDF files, you must have a current version of the Adobe Reader®, which you can download from Adobe at: www.adobe.com/products/acrobat/readstep2.html.

Table of Contents

1

Introduction

This chapter introduces the Neuron assembly language and the Neuron Assembler.

Introduction

An application program for a LONWORKS device that runs on a Series 3100 or Series 5000 Neuron Chip or Smart Transceiver uses the Neuron C programming language. Although this language is very powerful and flexible, there can be times when you want to optimize the application program for the LONWORKS device, perhaps for code size or processing speed. You can use Neuron assembly language to write functions or programs that provide such optimizations.

Although Neuron assembly language functions can be smaller and faster than those generated by the Neuron C compiler, they also have the following characteristics:

- The NodeBuilder FX Development Tool does not provide a Code Wizard for assembly functions
- There are fewer automated validations and checks for Neuron assembly code
- Functions written in assembly language can be harder to write, read, and maintain
- Functions written in assembly language have a larger potential for error, compared to higher language implementations
- Code written in assembly language cannot be debugged with the NodeBuilder Debugger

Nonetheless, the Neuron assembly language is a powerful tool for managing specific tasks for a Neuron C application program.

This chapter provides an overview of the tools, files, and syntax for Neuron assembly language functions. The rest of this book contains the following information:

- Chapter 2, *Neuron Architecture for Neuron Assembly Programming*, on page 15, provides an overview of the Neuron architecture, hardware resources, and addressing modes.
- Chapter 3, *Writing a Neuron Assembly Utility Function*, on page 35, provides an overview of stack-oriented programming and information about designing assembly language functions. It also describes a recommended approach to documenting changes to the stack.
- Chapter 4, *Interfacing with a Neuron C Application*, on page 47, describes how a Neuron assembly language function can work with a Neuron C application.
- Chapter 5, *Exploring an Example Function in Neuron Assembly*, on page 55, describes a simple example function in Neuron assembly.
- Chapter 6, *Neuron Assembly Language Instruction Statements*, on page 59, describes all of the supported Neuron assembly language instructions.
- Chapter 7, *Neuron Assembler Directives*, on page 131, describes the supported Neuron Assembler directives.

- Chapter 8, *System-Provided Functions*, on page 159, describes system-provided functions for various arithmetical or logical operations or for stack management.

This book also contains several appendixes with additional information.

Neuron Assembler Tools

You can create and edit Neuron assembly language files using any text editor, such as Windows Notepad. The primary tool for working with Neuron assembly language files is the Neuron Assembler. The Neuron Assembler translates your source code, written in the Neuron Assembly language, into a Neuron object file (**.no** extension). You can use the Neuron Librarian to create or manage code libraries of Neuron object files, including those created from assembly language files.

In general, you do not need to use the Neuron Assembler when creating a small number of utility functions for use with one specific Neuron C application. To use assembly source code within a Neuron C application, the Neuron C Compiler supports the **#pragma include_assembly_file** directive, which can be used to copy a specified assembly source file directly into the compiler-generated output.

The two main tools for working with Neuron C applications, which can interact with Neuron assembly functions, are the NodeBuilder FX Development Tool and the Mini FX Evaluation Kit. The NodeBuilder FX Development Tool can produce Neuron assembly listing output files for your Neuron C programs. However, the Mini FX Evaluation Kit does not produce Neuron assembly listing output files.

Neuron C Compiler for Assembly Programming

The Neuron C Compiler supports the **#pragma include_assembly_file** directive. This directive can be used repeatedly within the same Neuron C source code, specifying one assembly source file at a time.

The Neuron C compiler translates your Neuron C source code into Neuron Assembly output. When it encounters the **#pragma include_assembly_file** directive, the compiler copies the content of the referenced assembly source file directly (without modification) into its own output stream.

See the *Neuron C Reference Guide* for more information about compiler directives.

Neuron Assembler Command Line Tool

The Neuron Assembler, available from the command line as **NAS.EXE**, translates source files written in the Neuron assembly language (typically using a **.ns** file extension) into Neuron object files (**.no** file extension), which can then be packaged into function libraries using the Neuron Librarian (**NLIB.EXE**). The resulting function libraries can then be provided to the Neuron Linker (**NLD.EXE**), and code that is contained in these libraries and referenced by the Neuron C source code is linked with the application.

To run the Neuron Assembler, open a Windows command prompt (**Start** → **Programs** → **Accessories** → **Command Prompt**), and enter the following command:

```
nas –switches file.ns
```

where *–switches* define any optional command-line switches (see *NAS Command Switches* on page 4) and *file.*ns specifies the source input file to assemble. The command-line tools are installed in the **LonWorks\bin** directory.

For example, the following command runs the Neuron Assembler, assembles the **abc.ns** source file, and generates an **abc.no** object file in the same directory:

```
NAS abc.ns
```

Command Usage

The following command usage notes apply to running the **nas** command:

- If no command switches or arguments follow the command name, the tool responds with usage hints and a list of available command switches.
- Most command switches come in two forms: A short form and a long form.

  The short form consists of a single, case-sensitive, character that identifies the command, and must be prefixed with a single forward slash '**/**' or a single dash '**-**'. Short command switches can be separated from their respective values with a single space or an equal sign. Short command switches do not require a separator; the value can follow the command identifier immediately.

  The long form consists of the verbose, case-sensitive, name of the command, and must be prefixed with a double dash '**- -**'. Long command switches require a separator, which can consist of a single space or an equal sign.

  Examples:
  Short form: `nas –l …`

  Long form: `nas --listing …`
- Multiple command switches can be separated by a single space.
- Commands of a Boolean type need not be followed by a value. In this case, the value **yes** is assumed. Possible values for Boolean commands are **yes**, **on**, **1**, **+**, **no**, **off**, **0**, **-** (a minus sign or dash).

  Examples:
  ```
  nas --listing=yes abc.ns
  nas --listing abc.ns
  ```
- Command switches can appear at any location within the command line or in any order (on separate lines) within a script.

NAS Command Switches

Table 1 on page 5 lists the available command switches for the **nas** command. All switches are optional.

Table 1. Command Switches for the **nas** Command

Command Switch		Description
Long Form	Short Form	
--autotrunc	-a	Auto-truncate literals for byte-immediate operations
--define	-d	Define a specified conditional-compilation symbol
--defloc		Location of an optional default command file
--file	-@	Include a command file
--headroom	-h	Report available memory headroom
--help	-?	Display usage hint for command
--laserjet	-j	Make the listing LaserJet landscape
--listfile	-L	Specify an explicit listing file (with -l)
--listing	-l	Produce an assembly listing output file
--mkscript		Generate a command script
--nodefaults		Disable processing of default command files
--outfile	-o	Specify an explicit output file
--search	-s	Add a path to the search list for include files
--silent		Suppress banner message display
--suboptimalwarning	-w	Display warnings for instructions that are not optimal size
--warning		Display specified value as a warning

Neuron Librarian Tool

A library is a collection of Neuron object files. Each object file contains one or more compiled ANSI C source files or assembled Neuron assembly source files. Members of a library are object files created by the Neuron Assembler or the Neuron C compiler. Although you use the Neuron C compiler to create object files for a library, you cannot include language constructs that are specific to Neuron C in these functions.

The Neuron Librarian, available from the command line as **NLIB.EXE**, allows you to create and manage libraries, or add and remove individual object files to and from an existing library.

See the *Neuron C Programmer's Guide* for more information about the Neuron Librarian.

NodeBuilder Development Tool

The NodeBuilder FX Development Tool is a hardware and software platform for developing applications for Neuron Chips and Smart Transceivers. With the NodeBuilder FX Development Tool, you can perform many tasks for developing LONWORKS devices, including: write and edit Neuron C code, generate Neuron C code for the device interface, compile and build your application, and debug your application.

The NodeBuilder FX Development Tool does not work directly with Neuron assembly language files. However, it can produce an assembly listing output file for your Neuron C programs, and it can embed your assembly source files into your application if it includes the **#pragma include_assembly_file** directive. The NodeBuilder FX Development tool can also provide Neuron function libraries (which can contain object files based on your assembly source) to the linker, thus making these assembly-coded functions available to your application.

Note that the NodeBuilder FX Development Tool does not directly support debugging of code written in assembly or of any code brought in from a function library.

See the *NodeBuilder FX User's Guide* for more information about the NodeBuilder FX Development Tool.

Assembler Files

The Neuron Assembler requires a single source input file, and produces one object output file, and optionally also produces a listing output file. The descriptions and requirements for native Neuron assembly files also apply to Neuron assembly source files that are used with the Neuron C Compiler's **#pragma include_assembly_file** directive.

The following sections describe these files.

Source Files

A source file (the input file) contains zero, one, or more lines of Neuron assembly source code. This source code can consist of Neuron assembly instructions or Neuron assembly directives.

The following sections describe the file naming convention and file format for source files.

Naming Convention

A file that contains Neuron assembly source can have any file name that is allowed by the operating system, but the use of the **.ns** file extension is recommended. If no file name extension is provided on the command line, the Neuron Assembler assumes an **.ns** extension.

A file that is used with the Neuron Librarian (NLIB) must follow the Windows 8.3 naming convention (for example, *filename.ext*, where *filename* is no more

than eight characters and *ext* is one of the allowable extensions for Neuron assembly files, such as **.ns** and **.no**). Spaces are not allowed in the file name or the extension.

File Format

Each line in an assembly source file is independent, with no line-continuation character or terminator character. An assembly source line can contain one of the following:

- A blank line (zero, one, or more whitespace characters (blanks, spaces, or tabs), followed by a newline or carriage-return character). The Neuron Assembler ignores blank lines.
- A comment. A comment starts with a semicolon character (**;**) and ends with a newline or carriage-return character. The Neuron Assembler ignores all characters after the semicolon.
- A label. A label is an identifier for an instruction, directive, or segment.
- An assembly instruction, with zero, one, or more arguments. The line with the instruction can begin with an optional label, and can also end with an optional comment.
- An assembler directive. The line with the directive can begin with an optional label (if the directive allows one), and can also end with an optional comment.

If the line contains a label, it must begin in the first column. If the line contains an instruction or directive without a label, the instruction or directive must begin beyond the first column. A comment can begin in any column, but always extends to the end of the line.

Spaces or tabs separate the label, the instruction mnemonic or directive, and the first argument (if any). Multiple arguments are separated by spaces, tabs, or commas, depending on the syntax of the particular instruction or directive.

Thus, the basic format for an assembly source line is:

```
label instruction operand ; comment
label DIRECTIVE argument ; comment
```

By convention, instruction mnemonics are specified in lower case and directives are specified in upper case.

Output Files

The Neuron Assembler produces the following output files:

- An object output file
- An optional listing output file

The *object output file* contains assembled code, ready for the Neuron Linker. Typically, the object output file has the same name as the input source file, but has an **.no** file extension.

The *listing output file* contains the source instructions, directives, and comments from the source input file, and includes the instruction opcodes and addressing information (including symbolic or segment-relative addresses that are resolved

by the linker). The listing output file also can include source instructions and directives from imported files. The listing output file is formatted for ease of printing or viewing online. Use the **--listing (-l)** command-line switch to generate a listing output file. Typically, the listing output file has the same name as the input source file, but has an **.nl** file extension.

For both the object output file and the listing output file, you can use the **--outfile (-o)** command-line switch of the **nas** command to rename these files. However, this switch does not allow you to redirect the files to another directory.

A listing output file consists of one or more pages of output. Each page begins with two header lines and a subhead line:

- The first header line contains information about the version of the Neuron Assembler and ends with the current page number. This line starts with a page break control character (form feed), that is used for printing control. This control character is usually represented by a special symbol when you view the assembly listing file with a file editor.
- The second header line contains the date and time that the listing file was created and ends with the name of the source input file.
- The subhead line is blank, unless you specify a subhead using the **SUBHEAD** directive (see *SUBHEAD (Listing Control)* on page 158).

The rest of a page of a listing output file contains assembly source lines with additional information in the left-hand columns:

- The first field is a four-digit hexadecimal number that represents the absolute or relative address of the line. If the assembly source line defines a label, the four-digit number is the value of the label. For relocatable segments, the value is relative to the beginning of the segment.
- The second field (and subsequent fields) is a two-digit or four-digit hexadecimal number that represents the opcodes or data bytes as assembled. If the field contains four zeros followed immediately by an asterisk (*****), the field's value cannot be determined at assembly time, but must be resolved at link time.

Assembly source lines that are skipped because of conditional assembly are also included in the listing output file. Lines that are skipped are marked with an exclamation mark (**!**) in the left-most column to designate that the line was not assembled.

General Neuron Assembly Syntax

The general Neuron assembly language syntax is:

```
label keyword operand1 operand2 ; comment
```

where:

- *label* is an optional identifier, followed by white space
- *keyword* is a reserved name for an assembly instruction or Assembler directive
- The operands *operand1* and *operand2* are optional. There can be zero, one, or two operands, depending on the specific instruction. When

present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or stack pointers, or are assumed to be data items declared in another part of the file. Depending on the instruction, two operands are separated by either white space or a comma, and depending on the addressing mode, they can be enclosed in square brackets (**[ ]**) or combined with other special characters.

- The terminating semicolon and comment are optional, but highly recommended for most assembly source lines. See *Documenting Changes to the Stack* on page 40 for additional comment recommendations.

The following sections provide additional information about these syntax elements.

Labels

A *label* is an identifier for an instruction, directive, or segment. The Neuron Assembler defines label values by their position in the source file relative to the instructions and directives. A relocatable label has no absolute value until link time.

A label can comprise either lower or upper case letters a..z or A..Z, the digits 0..9, and any of the following special characters: underscore (**_**), period (**.**), or percent (**%**). The first character cannot be one of the numeric digits, nor can it be the period (**.**) character. Labels are case sensitive.

Note that Neuron Assembler keywords are reserved and cannot be used for labels, regardless of case. Reserved words include instruction mnemonics, assembler directives, or register names. See Appendix C, *Reserved Keywords*, on page 219, for a list of assembler keywords.

Assembly Instructions

An *assembly instruction* is a keyword that represents a class of machine operation codes (opcodes). The specific opcode is defined by the instruction, combined with its operands.

An assembly instruction can comprise either upper or lower case characters a..z or A..Z and the underscore character (**_**). Assembly instructions are not case sensitive.

Each instruction accepts zero, one, or two operands (which can include special characters).

An exclamation character (**!**) can precede an instruction. This character indicates that any warning that results from using the **-w** command line switch is suppressed for that instruction. Using this character can be useful if the **-w** switch warns of something that is intentional or otherwise unavoidable.

See Chapter 6, *Neuron Assembly Language Instruction Statements*, on page 59, for a description of all supported instructions.

Operands

Many Neuron assembly instructions require one or two operands to define the machine instruction (opcode). Operands add information to the instruction, and define the data that the instruction should operate on.

For example, some instructions require a register name (such as TOS or DSP) as an argument to specify the source or destination of the data for the instruction. In general, you can specify operand names in upper or lower case.

Some instructions use immediate addressing, for which you specify the operand by prefixing a number sign or hash (**#**) to the value. An immediate value is used as a literal value. For example, a **PUSH #24** instruction pushes the literal value "24" onto the stack, whereas a **PUSH 24** instruction pushes the contents of location 24 onto the stack.

Some instructions use base-relative addressing, for which the operand specifies a location within the base-page relative to its starting address. Specify such a displacement by prefixing an exclamation mark (**!**) to the operand.

Some instructions use one or two operands that specify a pointer register, sometimes also with a displacement, or they specify a displacement relative to the data stack pointer (DSP) or return stack pointer (RSP). Specify these types of arguments by enclosing the argument in square brackets (**[ ]**).

Literal Constants

A *literal constant* is a numeric value, such as 12 or 173. The Neuron Assembler supports numeric values in any of four radixes (bases): binary (base 2), octal (base 8), decimal (base 10), and hexadecimal (base 16).

The **RADIX** directive specifies the default radix for an assembly language function (see *RADIX (Default Radix)* on page 154). To explicitly specify the radix for a literal constant, prefix the constant's value with one of the following letters and the apostrophe character (**'**):

- b' for binary numbers
- o' for octal numbers
- d' for decimal numbers
- h' for hexadecimal numbers

You can specify the radix letter in either upper or lower case. You can specify leading zeros for any literal constant without changing its value. However, if the default radix is hexadecimal, a literal constant must always begin with a numeric digit or leading zero.

For example, you can specify the decimal value 123 as b'01111011, o'173, d'123, or h'7b.

Symbols

A *symbol* is one or more consecutive alphanumeric characters. A symbol can comprise either lower or upper case letters a..z or A..Z, the digits 0..9, and any of the following special characters: underscore (**_**), period (**.**), or percent (**%**). The

first character cannot be one of the numeric digits, nor can it be the period (.) character. Symbols are case sensitive.

Note that Neuron Assembler keywords are reserved and cannot be used for symbols, regardless of case. Reserved words include instruction mnemonics, assembler directives, or register names. See Appendix C, *Reserved Keywords*, on page 219, for a list of assembler keywords.

A label is a type of symbol (see *Labels* on page 9). A symbol that acts as a label for the **EQU** directive is defined explicitly by the directive's argument expression, regardless of whether the directive is in a relocatable segment. See *EQU (Equate Symbol)* on page 140 for more information about this directive.

You can also define a symbol by importing its value from an assembled object file. The value of an imported symbol is known only at link time. In addition, you can export a symbol to make it available at link time to another assembly file or a Neuron C file. See one of the following sections for more information about importing and exporting symbols: *APEXP (Application Symbol Export)* on page 134, *EXPORT (Export Symbol)* on page 142, and *IMPORT (Import External Symbol)* on page 146.

Expressions

Some Neuron assembly instructions accept expressions as arguments. The simplest *expression* consists of a literal constant or a symbol. However, the assembler also accepts expressions for which the value is the result of a computation. The computation can involve multiple literal constants or symbols and use a variety of operators. If the value of an expression is not computable at assembly time, the assembly object output file must contain sufficient information for the Neuron Linker to compute the expression value at link time.

General Expressions

The assembler supports the following types of operators for creating general expressions:

- Unary
- Binary
- Special operators

All expression values are 16-bit expressions, and all operators produce 16-bit results. All operations use unsigned two's-complement arithmetic. You can add parentheses to an expression to syntactically determine its boundaries without changing its value.

Unary operators are symbols that appear in front of an expression and perform an operation on the value of that expression. **Table 2** lists the unary operators.

Table 2. Unary Operators

Operator	Description
-	Two's-complement arithmetic negation
~	One's-complement bitwise negation

Binary operators are symbols that appear between two expressions and perform an operation to combine the values of the two expressions. **Table 3** lists the binary operators.

Table 3. Binary Operators

Operator	Description
+	Two's-complement addition
-	Two's-complement subtraction
*	Multiplication
/	Division with integral truncation
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR

The *special operators* are symbols that instruct the Neuron Assembler to perform a specific action. The values of expressions that use special operators are computed at link time. **Table 4** lists the special operators.

Table 4. Special Operators

Operator	Description
@	Specifies a special operator function
*	Specifies the absolute address of the current assembly instruction (only computable at link time for relocatable segments)

A special operator function begins with the @ character, followed by the function name, and then by a parenthesized expression. The function names can be specified in either lower or upper case. **Table 5** lists the special operator functions.

Table 5. Special Operator Functions

Function	Description
LB	Extracts the low byte of the expression
HB	Extracts the high byte of the expression (logical shift right by eight bits)
NEAR	Offset value for the RAMNEAR area (only computable at link time)

Example:

The following example demonstrates the use of the @NEAR expression. The example implements a one-byte variable in the RAMNEAR segment, and a function named *Example* that increments this global variable.

```
; open RAMNEAR segment, declare one byte
; uninitialized variable

         SEG RAMNEAR
         ORG
myVariable EXPORT
         RES 1

; The Example routine takes no arguments and produces
; no results, but increments the global variable

         SEG CODE
         ORG

pNear    EQU  1

Example  APEXP               ; ( -- )
         push [pNear][@NEAR(myVariable)]
         inc
         pop  [pNear][@NEAR(myVariable)]
         ret                 ; return to caller
```

Constant Expressions

A *constant expression* is a general expression that is computable at assembly time and has a constant value in an appropriate range for the specified instruction or directive.

An expression that is not constant is one that is computable only at link time or that has a variable value at runtime. Such symbols must be designated as imported symbols.

Important: An expression that uses either the * or @NEAR special operator cannot be a constant expression.

When using negative constants, or expressions that yield negative results, in a byte context, you must use the @LB special operator (or specify the **--autotrunc** [**-a**] command-line switch) to obtain a one-byte value for the value. For example, use **PUSH #@LB(-1)** instead of **PUSH #-1** (the latter will fail assembly).

Displacements

A *displacement* is a relative address value. A displacement can be interpreted as a signed or unsigned number, depending on the instruction with which the displacement appears.

To compute the absolute address value, add the displacement (using either signed or unsigned arithmetic, as appropriate) to the absolute address of the instruction that contains the displacement.

Address Expressions

An *address expression* is an expression that specifies an address, and can be one of the following types of expression: a literal constant, a symbol with an optional offset expression, or the * special operator with an optional offset expression. The optional offset expression is a general expression with a prefixed + (addition) or - (subtraction) operator.

An address expression can consist of a mixture of locally defined and imported symbols from the same or from multiple segments. However, symbols from other segments must be exported, even if they are not used by other modules.

Comments

A comment is part of an assembly source line that provides useful information to the code developer. The Neuron Assembler ignores comments. A comment starts with a semicolon character (;) and ends with a newline or carriage-return character.

Recommendation: Use comments to document changes to the data and return stacks. See *Documenting Changes to the Stack* on page 40 for a recommended method of documenting stack changes.

Assembler Directives

An assembler directive provides information to the Neuron Assembler to control or affect the processing of the remainder of the assembly file.

The directives have syntax analogous to the assembly instructions. Most directives require arguments, which are similar to the operands of the assembly instructions.

See Chapter 7, *Neuron Assembler Directives*, on page 131, for more information about the Neuron Assembler directives.

Interfacing with Neuron C Programs

Typically, you use Neuron assembly language to create utilities that can be used with an application that is written in Neuron C. The Neuron C program code might call functions defined in Neuron assembly, or a Neuron assembly function might call a Neuron C function or program.

See Chapter 4, *Interfacing with a Neuron C Application*, on page 47, for more information about how Neuron assembly language functions and Neuron C program interact.

2

Neuron Architecture for Neuron Assembly Programming

This chapter describes elements of the Neuron architecture that apply to writing a function in Neuron assembly language.

Neuron Architecture

For Series 3100 devices, the architecture of a Neuron Chip or Smart Transceiver includes three independent processors that share a common memory, arithmetic-logic unit (ALU), and control circuitry. Each processor has its own set of registers, including an instruction pointer (IP) and a flag register (FLAGS), which contains the processor ID and the Carry flag.

For Series 5000 devices, the architecture of a Neuron Chip or Smart Transceiver is essentially identical to the independent processors of a Series 3100 device; however, Series 5000 devices also provide interrupt-processing support. Depending on the device's configuration, interrupts can run in a fourth processor or share the main application processor. As with Series 3100 devices, each processor has its own set of registers, including an instruction pointer (IP) and a flag register (FLAGS), which contains the processor ID and the Carry flag.

The Neuron architecture uses a *base page* model for addressing memory. Each processor has a base-page register (BP) that points to a 16-byte boundary in RAM. The first eight bytes of the base page are used as four 16-bit pointers (named P0 to P3), followed by 16 bytes that implement 16 one-byte data registers (named R0 to R15).

Some addressing modes refer to those general-purpose registers, and one addressing mode directly accesses all 256 base-page bytes, including the 24 bytes for the general-purpose registers.

Many of the general-purpose registers have a pre-defined use within the Neuron system firmware and application framework. See *CPU Registers* on page 17 for more information.

Base-page pointers or the direct addressing mode is used to access to global data (defined as memory outside the base page).

Neuron Chips and Smart Transceivers are stack-oriented machines, using two stacks: the data stack and the return stack. The data stack holds program data, and the return stack holds return addresses and transient local data. In the event of an interrupt, the return stack also holds some of the processor's state information.

The data stack's starting address is at low base page memory (after the general-purpose register area), and moves upward. The data stack pointer (DSP) is an 8-bit offset from the BP register. The return stack's starting address is at the top of the base-page memory, and moves downward. The return stack pointer (RSP) is an 8-bit offset from the BP register.

A dedicated hardware register holds the top of data stack element (this element is called TOS), and a special addressing mode allows for fast access to the element below TOS (this element is called NEXT).

This stack-oriented architecture, with both data and return stacks growing towards each other within the same 256-byte base page, is not normally problematic, but deep recursion or large local variables, or a combination of both, should be avoided to prevent the stacks from colliding. The programmer is responsible for seeing that the two stacks never collide. For Series 5000 devices, a stack collision or a stack underflow can be recognized by the system firmware, and results in an entry in the device's error log.

A Neuron Chip or Smart Transceiver is a big-endian device, that is, the most-significant byte (MSB) of an address or a 16-bit scalar is at a lower memory address, and the least-significant byte (LSB) of an address or a 16-bit scalar is at a higher memory address. For 16-bit addresses, a Neuron assembly language function must be sure to read or write the MSB at a low address before reading or writing the LSB at higher address.

Because the data stack grows towards higher addresses, 16-bit entities appear on the data stack with the LSB nearer to TOS. For the return stack, which grows in the opposite direction, 16-bit entities appear with the MSB nearer to TOS.

Most operations that require arguments require that these arguments are pushed onto a stack, and when an operation is performed, its arguments are popped from a stack and its result (if any) pushed back on.

In addition to the DSP, a Neuron assembly language function can use two registers, TOS and NEXT, to work with the top of the stack and with the next element below the top of the stack, respectively. See *Overview of Stack-Oriented Programming* on page 36 for additional information about working with stacks.

For more information about the Neuron architecture:

- For Series 3100 devices, see the *FT 3120 / FT 3150 Smart Transceiver Data Book* or the *PL 3120 / PL 3150 / PL 3170 Power Line Smart Transceiver Data Book*.
- For Series 5000 devices, see the *Series 5000 Chip Data Book*.

Hardware Resources for Assembly Programs

The Neuron architecture provides two major sets of hardware resources for a Neuron assembly language function to use: CPU registers and stacks. To make effective use of these resources, the Neuron assembly language implements a number of addressing modes that provide efficient access to data, either through the registers or through one of the stacks. The following sections describe these hardware resources. See *Addressing Modes* on page 30 for a description of the available addressing modes.

CPU Registers

The Neuron architecture provides the following types of CPU registers for Neuron assembly language programming:

- General-purpose 16-bit pointer registers and 8-bit data registers
- A flag register
- An instruction pointer
- A base-page register
- A data-stack pointer register (8-bit, BP-relative)
- A register containing the element on top of the data stack (TOS)
- A return-stack pointer register (8-bit, BP-relative)

The following sections describe these registers.

General-Purpose Registers

The Neuron architecture defines 16 hardware memory locations that the Neuron Assembler uses as general-purpose registers, typically named R0, R1, R2, and so on, to R15, as described in **Table 6**. Each of the general-purpose registers is eight bits wide.

Table 6. General-Purpose Registers

General-Purpose Register	Description
R0	Scratch register. A function can use this register as needed. Assume that any call to the firmware, or to any function in a library or written in Neuron C, might change the contents of this register.
R1	Scratch register. A function can use this register as needed. Assume that any call to the firmware, or to any function in a library or written in Neuron C, might change the contents of this register.
R2	Scratch register. A function can use this register as needed. Assume that any call to the firmware, or to any function in a library or written in Neuron C, might change the contents of this register.
R3	Reserved for use by the system firmware.
R4	Reserved for use by the system firmware.
R5	Reserved for use by the system firmware.
R6	Reserved for use by the system firmware.
R7	Reserved for use by the system firmware.
R8	Reserved for use by the system firmware.
R9	Reserved for use by the system firmware.
R10	Reserved for use by the system firmware.
R11	Reserved for use by the system firmware.
R12	Reserved for use by the system firmware.
R13	Reserved for use by the system firmware.
R14	Reserved for use by the system firmware.
R15	Reserved for use by the system firmware.

A scratch register is one that can be used for temporary data storage by multiple programs or functions. Although some operations support a special addressing mode to index data through the general-purpose registers, you should consider registers R0..R2 as modified after calling any other function. See Chapter 8, *System-Provided Functions*, on page 159, for a description of these functions, including which general-purpose registers each function uses and modifies.

Important: Do not use or modify R4 in your Neuron assembly functions because the network processor uses it. Modifying the other reserved general-purpose registers (R3 to R15) can also cause unpredictable results.

The Neuron Assembly language refers to the general purpose data registers by their base-page relative indices, 8..23. A typical assembly language function defines the mnemonics R0 to R15 for these registers by using the **EQU** assembly directive.

Example: The following **push** instruction uses base page relative addressing. This addressing mode addresses the argument through one operand, the index of one of the general purpose data registers within the base page. This index must be in the 8..23 range.

The following instruction pushes the value of the R0 register onto the data stack:

```
        push    !8
```

Defining mnemonic names for the general-purpose registers makes this instruction more easily readable. The following example is equivalent to the previous one:

```
R0      EQU     8
        push    !R0
```

However, because you define the R0 mnemonic, it carries no special meaning for the Neuron Assembler. The Assembler does not validate that the mnemonic refers to the correct or intended register.

For the purpose of clarity (and unless explicitly mentioned to the contrary), all source code examples in this book assume that the following mnemonics are defined:

```
R0      EQU     8
R1      EQU     R0+1
...
R15     EQU     R14+1
```

Finally, note that the mnemonics are user-defined symbols. Unlike pre-defined register names or assembly instructions, user-defined symbols are case-sensitive.

Pointer Registers

The Neuron architecture defines four general-purpose pointer registers, typically named P0, P1, P2, and P3, as described in **Table 7** on page 20. Each of the pointer registers is 16 bits wide.

Table 7. Pointer Registers

Pointer Register	Description
P0	Scratch register. A function can use this register as needed. Assume that any call to the firmware, or to any function in a library or written in Neuron C, might change the contents of this register.
P1	Always points to the beginning of the RAMNEAR segment. Do not change this register's content when working in a Neuron C context.
P2	Used by Neuron C programs. Assume that any call to the firmware, or to any function in a library or written in Neuron C, might change the contents of this register.
P3	Scratch register. Assume that any call to the firmware, or to any function in a library or written in Neuron C, might change the contents of this register.

A scratch register is one that can be used for temporary data storage by multiple programs or functions. That is, must typically be saved before and restored after calling any other function.

Important: Do not modify P1 or P2 in your Neuron assembly functions.

The Neuron Assembly language refers to the general-purpose pointer registers by their numerical identifier, 0..3. A typical assembly language function defines the mnemonics P0 to P3 for these registers by using the **EQU** assembly directive.

Example: The following **push** instruction uses indirect-relative addressing. This addressing mode addresses the argument through two operands: a pointer register and an offset. The effective address of the operation's argument is obtained by adding the offset to the contents of the pointer register.

The following instruction pushes the 7th byte from the memory address pointed to by the P0 register onto the data stack:

```
        push   [0][7]
```

Defining mnemonic names for the general-purpose registers makes this instruction more easily readable. The following example is equivalent to the previous one:

```
P0      EQU    0
        push   [P0][7]
```

However, because you define the P0 mnemonic, it carries no special meaning for the Neuron Assembler. The Assembler does not validate that the mnemonic refers to the correct or intended register.

For the purpose of clarity (and unless explicitly mentioned to the contrary), all source code examples in this book assume that the following mnemonics are defined:

```
P0      EQU    0
```

```
P1      EQU     1
P2      EQU     2
P3      EQU     3
```

Finally, note that the mnemonics are user-defined symbols. Unlike pre-defined register names or assembly instructions, user-defined symbols are case-sensitive.

Flag Register

The flag register (FLAGS) is an 8-bit data register that contains the Carry flag and other flags that are used by the system firmware. The Carry flag is the most-significant bit of register, as shown in **Figure 1**.

A Neuron assembly function would rarely use the FLAGS register explicitly; however, a number of assembly instructions for conditional branches and arithmetic use the Carry flag from this register.

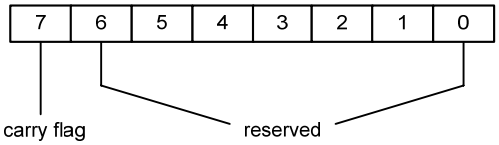

Figure 1. Flag Register Bits

Instruction Pointer

The instruction pointer (IP) is a 16-bit address register that contains the address of the next instruction to be executed. It is explicitly modified by call, branch, and return instructions. Conditional branch instructions (when the branch is not taken) and all other Neuron assembly instructions implicitly increment the IP by the size of the instruction.

Base-Page Register and Stack Registers

Each of the Neuron processors has its own base page area in the data space. The base pages are established at device startup by an initialization function. Any remaining data space that is not used for base pages can be used as global variable space.

A base page area has four parts:

- General-purpose pointer registers
- General-purpose data registers
- Data stack and data-stack register
- Return stack and return-stack register

Figure 2 on page 22 shows an example for the layout of a base page, including all of the registers that are available for Neuron assembly language programming.

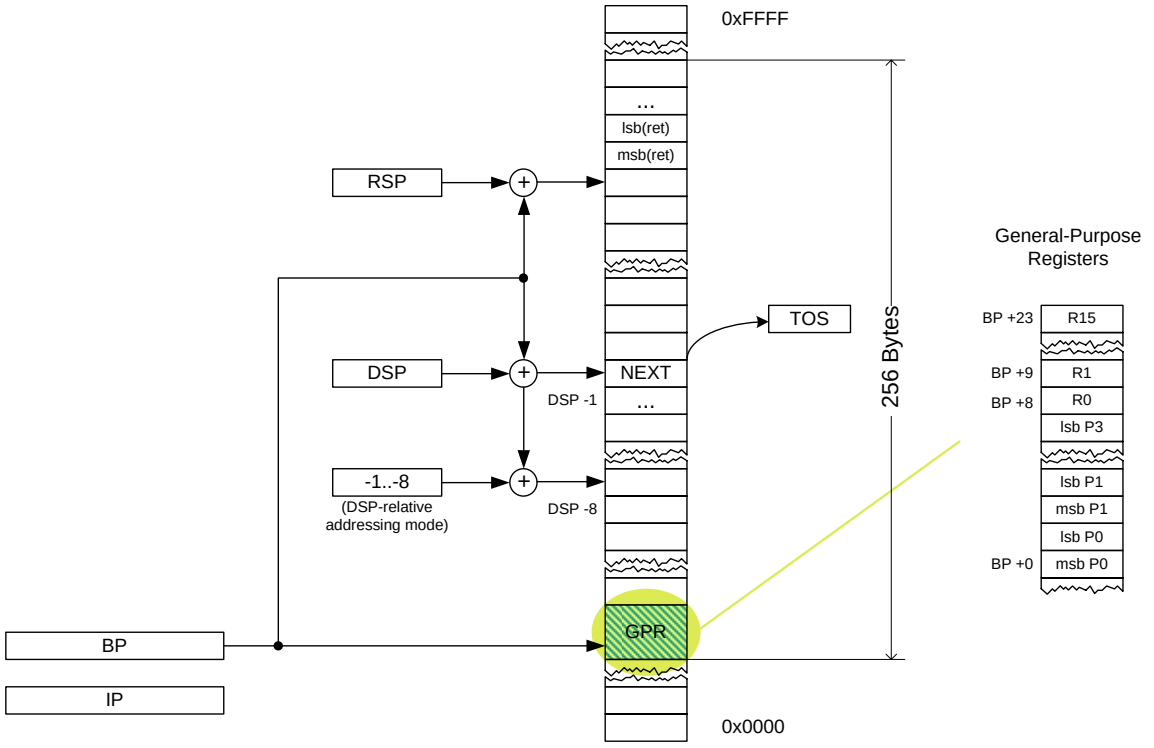

Figure 2. Base Page Layout

Stacks

Each of the Neuron processors has two stacks: a data stack and a return stack. The data stack starts at lower addresses and grows toward high memory. The return stack starts at the top of the base page and grows toward low memory. In general, it is up to the programmer to ensure that these two stacks do not overlap.

Data Stack

The data stack is located both in registers and in base page RAM, that is, it consists of two registers and associated memory. Access to the data stack is controlled by two registers:

- Top of stack (TOS) is an 8-bit data register that holds the top element of the data stack.
- Data stack register (DSP) is an 8-bit address register that, when added to the BP, points to the location in the data stack where the NEXT element exists. The DSP grows towards higher addresses.

Example: If the values 1, 2, 3, and 4 were on the data stack, and a **PUSH** instruction pushed the value 5 onto the data stack, the register contents would be as shown in **Figure 3** on page 23. Note that DSP points to the NEXT element.

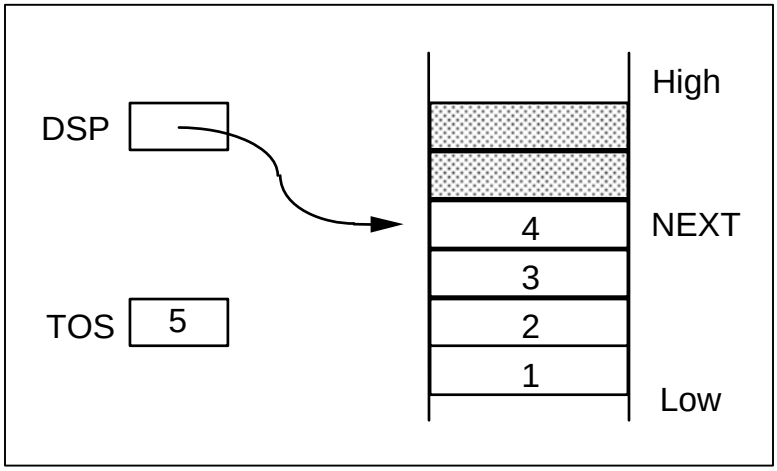

Figure 3. The Data Stack after a Push

For two-byte quantities, a two-byte push places the most-significant byte (MSB) onto the stack first, followed by the least-significant byte (LSB).

Some assembly instructions support the DSP-relative addressing mode, which provides easy access to the eight bytes of data in the stack below NEXT (expressed with their negative DSP-relative offset -1..-8).

Return Stack

The return stack is also located in base page RAM. It consists of one register and associated memory. The return stack pointer (RSP) is an 8-bit address register that is added to the BP to address the top of the return stack. The RSP grows towards lower addresses.

When calls are executed, the address of the next sequential instruction is stored on the top of the return stack. This is a 16-bit address stored with the least significant byte first (that is, the high order byte is pushed last).

Example: If a call were executed from location H'0124, the contents of the return stack would be as shown in **Figure 4** on page 24.

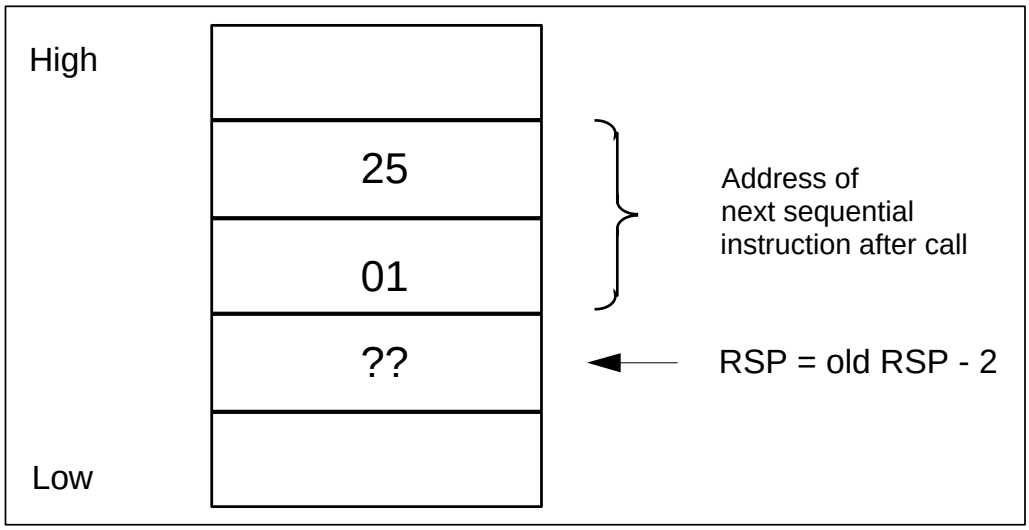

Figure 4. The Return Stack

Stack Manipulation

The Neuron assembly language provides the following instructions to manipulate the data stack:

- **PUSH** to place data from literal data, an address, or a register into the TOS element of the stack
- **PUSH TOS** to copy the value in the TOS element of the stack into the TOS element (so that it occupies both the TOS and NEXT elements)
- **PUSH NEXT** to copy the value in the NEXT element to the TOS element
- **POP** to remove data from the TOS element of the stack and place it into an address or a register
- **ALLOC** to allocate space on the stack
- **DEALLOC** to deallocate space on the stack
- **DROP** to delete data from the TOS element of the stack or from the NEXT element of the stack
- **INC** and **DEC** to increment and decrement the TOS element on the stack
- **ROLC** and **RORC** to rotate the value in the TOS element on the stack left or right through the Carry flag
- **SHL**, **SHLA**, **SHR**, and **SHRA** to shift the value in the TOS element on the stack left or right, either logically or arithmetically
- **XCH** to exchange the value in the TOS element of the stack with the value in the NEXT element of the stack

In addition, the Neuron assembly language provides the following instructions to work with both the data stack and the return stack:

- **PUSHPOP** to remove the top element of the return stack and place it into the TOS element of the data stack

- **POPPUSH** to remove the TOS element of the data stack and place it onto the top of the return stack
- **DROP [RSP]** to drop the top of the return stack
- **PUSH RSP** and **PUSH [RSP]** to copy the value from the top of the return stack into the TOS element of the data stack
- **POP RSP** to remove data from the TOS element of the data stack and place it onto the top of the return stack

See Chapter 6, *Neuron Assembly Language Instruction Statements*, on page 59, for more information about these and other Neuron assembly instructions.

Segments

The assembler groups assembly instructions and data blocks into one or more segments. A *segment* is a group of assembly instructions and data blocks that are assembled into consecutive bytes of machine instructions and data, at consecutive and ascending addresses. That is, when a segment is closed, it is complete and can be assembled and written to the object output file.

The basic unit for the assembler is a statement, but the basic unit for the linker is a segment. The linker relocates code and data on a segment-by-segment basis.

During assembly:

- One segment of each type is always open.
- Additional code and data can be added only to an open segment.
- Only one of the open segments is the currently active segment.
- The next assembly instruction or data statement encountered in the assembly file is added to the active segment.

The segment type does not affect the assembly of the source lines within it, but does affect the linking of the segment.

The Neuron Assembler supports the segment types listed in **Table 8**.

Table 8. Segment Types

Segment Name	Usage
CODE	For general assembly code.
EECODE	For code and constant data that resides in on-chip or off-chip EEPROM.
EEFAR	For data that resides in off-chip non-volatile memory.
EENEAR	For data that resides in on-chip EEPROM for Series 3100 devices or mandatory EEPROM for Series 5000 devices. The size of the EENEAR segment is 256 bytes.
INITCODE	For initialization code. Used by the Neuron C compiler.
INITDATA	For initialization data. Used by the Neuron C compiler.

Segment Name	Usage
RAMCODE	For code and constant data that resides in off-chip RAM.
RAMFAR	For data that resides in on-chip or off-chip RAM.
RAMNEAR	For data that resides in on-chip or off-chip RAM. Only Neuron 3150 Chip, FT 3150 Smart Transceiver, and Series 5000 devices support off-chip RAM. The size of the RAMNEAR segment is 256 bytes.
ROM	For code or constant data that resides in the user area of ROM. Only Neuron 3150 Chip and FT 3150 Smart Transceiver support a ROM user area.
SIDATA	For self-identification data. Used by the Neuron C compiler.

Notes:

- EEPROM: electrically erasable programmable read-only memory
- RAM: random access memory
- ROM: read-only memory

Using Neuron Chip Memory

The following sections describe the memory maps for Neuron Chips or Smart Transceivers with off-chip memory, and Neuron Chips or Smart Transceivers without off-chip memory. These memory maps show how several of the segment types relate to memory usage within the Neuron Chip or Smart Transceiver.

Chips with Off-Chip Memory

On-chip memory on the Neuron 3150 Chip and FT 3150 Smart Transceiver consists of RAM and EEPROM. On-chip memory on a Series 5000 device consists of RAM.

For the Neuron 3150 Chip and the FT 3150 Smart Transceiver, off-chip memory on these chips consists of one or more of ROM, RAM, EEPROM, NVRAM, or flash memory regions. You specify the starting page number for each region and the number of pages (a page is 256 bytes) when the device is defined. If ROM is used, its starting address must be 0000. If ROM is not used, then flash or NVRAM memory must take its place, starting at address 0000. The regions of memory must be in the order shown in **Figure 5** on page 27. They need not be contiguous, but they cannot overlap.

For Series 5000 devices, off-chip non-volatile memory can be either EEPROM or flash memory. Off-chip non-volatile memory starts at address 0x4000, and can extend up to address 0xE7FF. This memory area is called extended non-volatile memory. **Figure 6** on page 28 shows the memory map for Series 5000 devices.

Memory mapped I/O devices can be connected to the Neuron 3150 Chip, FT 3150 Smart Transceiver, or PL 3150 Smart Transceiver. The devices should respond only to memory addresses that correspond to any of the shaded areas in **Figure 5**.

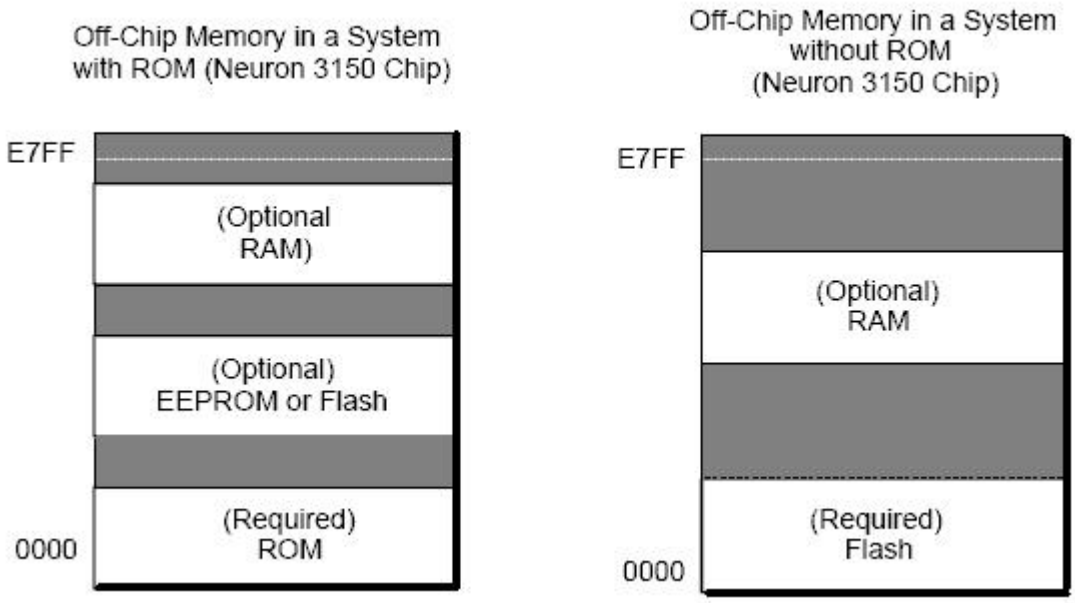

Figure 5. Off-Chip Memory on the Neuron 3150 Chip, the FT 3150 Smart Transceiver, and PL 3150 Smart Transceiver

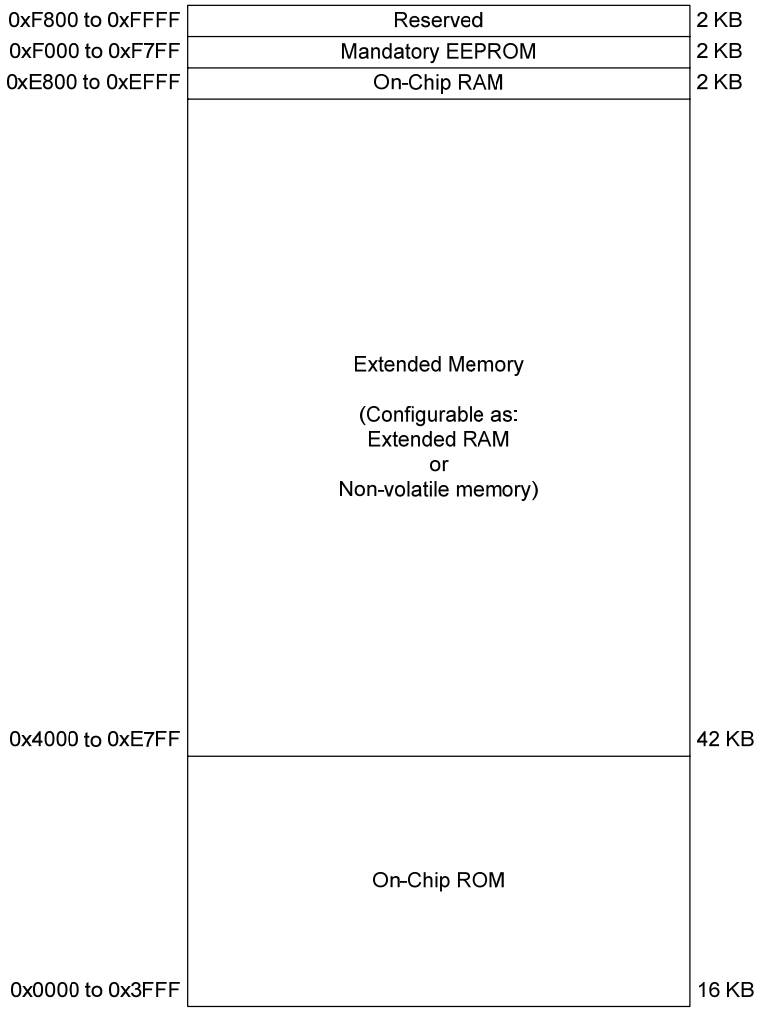

Figure 6. Memory Map for Series 5000 Devices

Chips without Off-Chip Memory

On-chip memory on the Neuron 3120 Chips, the 3120 Smart Transceiver, and PL 3170 Smart Transceiver consists of ROM, RAM, and EEPROM. None of these devices support off-chip memory. **Figure 7** on page 29 summarizes the memory maps.

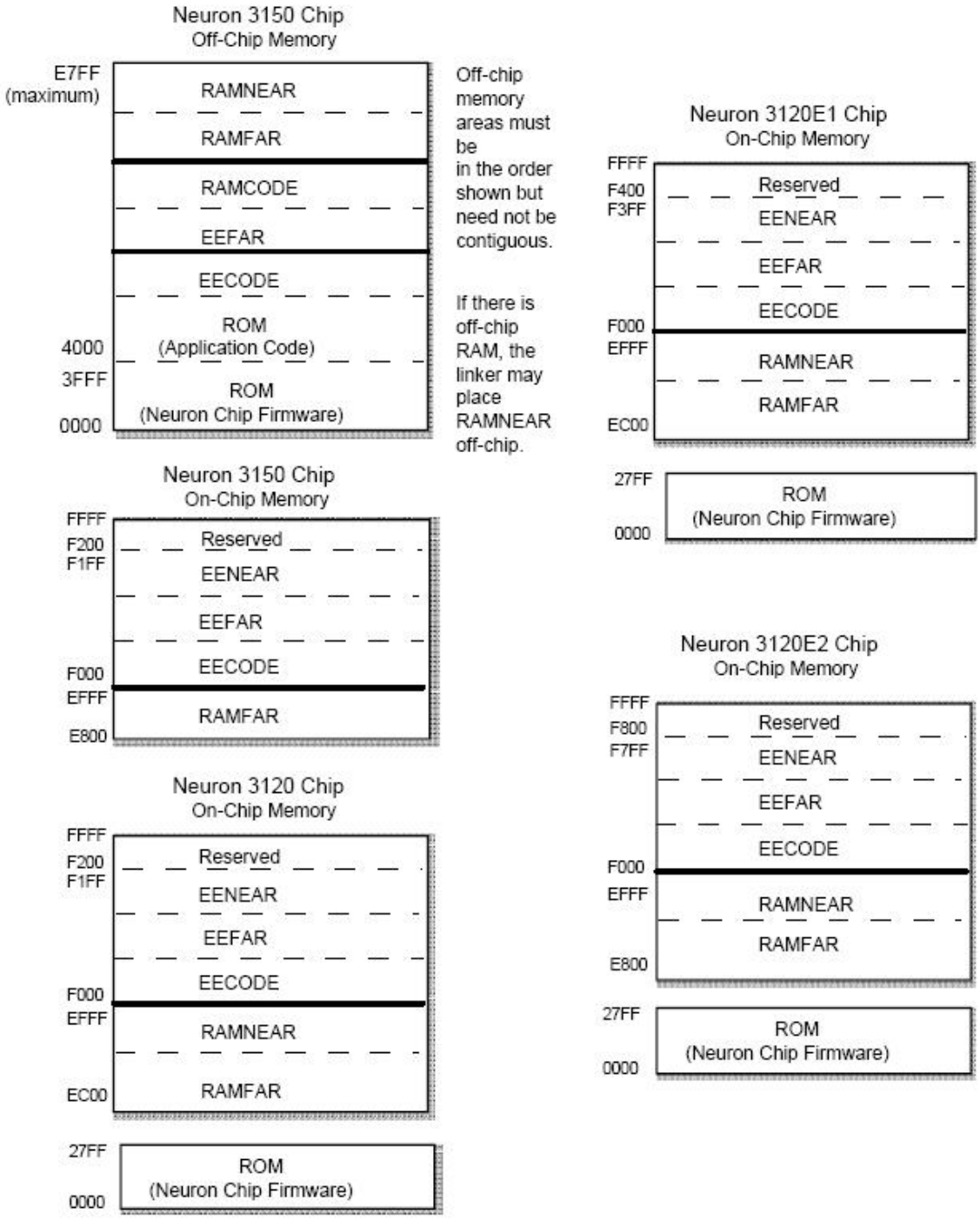

Figure 7. Memory Maps for Neuron Chips without Off-Chip Memory

Accessing Global and Static Data

The Neuron C compiler places each global and static data object in its own relocatable data segment of the appropriate type (RAMNEAR, RAMFAR, EENEAR, EEFAR, or ROM, depending on the storage type declared for the

Neuron C variable). For a Neuron assembly language function, you should also place global and static data into relocatable data segments.

Addressing Modes

The Neuron assembly language supports several addressing modes to specify the effective address of the argument of an instruction. The syntax for the instruction operand generally specifies the addressing mode used. For example, the number sign or hash (**#**) denotes the immediate addressing mode.

Not all of the addressing modes are available for all instructions. See Chapter 6, *Neuron Assembly Language Instruction Statements*, on page 59, for a description of the instructions and the addressing modes that each supports.

The following sections describe the supported addressing modes.

Immediate

In the immediate addressing mode, the instruction operand is a literal constant or an expression that resolves to a constant value. In this addressing mode, the instruction does not reference any registers or memory addresses.

Example:

```
push  #10           ; (10, ...)
pushd #h'98bc       ; (bc, 98, ...)
```

The **push #10** instruction pushes the number 10 onto the data stack. The **pushd #h'98bc** instruction pushes the most-significant byte (h'98) into NEXT and the least-significant byte (h'bc) into TOS.

Absolute

In the absolute addressing mode, the instruction operand is an absolute address in memory or an expression that resolves to an absolute address. Because the address is an absolute address, the address must be 16 bits.

Example:

```
push  myVariable    ; (myVar, ...)
callf myRoutine     ; (--)
```

The **push myVariable** instruction pushes the value at the location named *myVariable* into TOS. The **callf myRoutine** instruction is a far call to the *myRoutine* function.

Direct

In the direct addressing mode, the instruction operand is a register name or an expression that resolves to a name of a register. In contrast with the relative addressing modes, the direct addressing mode manipulates the contents of the register, rather than using the register as a pointer or address.

Example:

```
push tos            ; (TOS, ...)
sub  tos,next       ; (TOS-NEXT, ...)
```

The **push tos** instruction pushes TOS onto the stack, thus duplicating the value in TOS. That is, both TOS and NEXT now hold the same value. The **sub tos,next** instruction subtracts NEXT from TOS, and places the result in TOS. The result of this two-instruction sequence is zero (in TOS).

Implicit

In the implicit addressing mode, the instruction takes no operands. The instruction's arguments are implicit.

Example:

```
add                 ; (TOS+NEXT, ...)
pushpop             ; ( )
```

The **add** instruction adds the contents of TOS with the contents of NEXT, and places the result in TOS. The **pushpop** instruction pops the top element from the return stack and pushes it onto the data stack.

Pointer Direct

In the pointer-direct addressing mode, the instruction operand is a pointer register (P0, P1, P2, or P3) or an expression that resolves to a pointer register. In contrast with the relative addressing modes, the pointer-direct addressing mode manipulates the contents of the register, rather than using the register as a pointer or address.

Example:

```
inc   [P0]          ; ( )
pushd [P3]          ; ([P3], ...)
```

The **inc [P0]** instruction increments the pointer register P0. The **pushd [P3]** instruction pushes the two-byte pointer register P3 onto the data stack.

Indirect Relative

In the indirect-relative addressing mode, the instruction takes two arguments: a pointer register (P0, P1, P2, or P3) or an expression that resolves to a pointer register, plus an offset value or expression that resolves to an offset value. The offset must be in the range 0 to 255. The indirect-relative addressing mode uses the pointer register plus offset as a pointer or address, rather than using the contents of the register.

Example:

```
push  [P0][12]       ; ( )
pop   [P3][4]        ; ([P3][4], ...)
```

The **push [P0][12]** instruction pushes the 12th byte from the address contained in the P0 general-purpose pointer register onto the data stack by adding the value of the second operand (12) to the value of the P0 pointer register. The **pop [P3][4]** instruction pops the value in TOS to the location specified as 4 bytes from the value of the P3 pointer register.

Indirect Indexed

In the indirect-indexed addressing mode, the instruction takes two operands: a pointer register (P0, P1, P2, or P3) or an expression that resolves to a pointer register, plus an offset value in TOS. Unlike the indirect-relative addressing mode, the offset can be any value. In contrast with the direct addressing modes, which uses constant offset values, this addressing mode uses variable offset values in the 0..255 range, contained in TOS.

Example:

```
pop  [P0][tos]      ; ( )
push [P3][tos]      ; ([P3][4], ...)
```

The **pop [P0][tos]** instruction pops the value of NEXT from the stack and places it into the location specified by adding the value of TOS to the value of the P0 pointer register. The **push [P3][tos]** instruction pushes the value from the location specified by adding TOS to the value of the P3 pointer register into NEXT on the stack.

DSP Relative

In the DSP-relative addressing mode, the instruction takes two operands: the data stack pointer (DSP) and a negative displacement value or expression that resolves to a negative displacement value. The displacement must be in the range -1 to -8.

Example:

```
                    ; (t, n, a, b, c, d, e --)
push [dsp][-1]      ; (a, t, n, a, b, c, d, e -- )
pop  [dsp][-4]      ; (t, n, a, b, c, a, e --)
```

The **push [dsp][-1]** instruction pushes the a copy of the first element below NEXT into TOS ('a' in this example). The **pop [dsp][-4]** instruction pops the value held in TOS into the fourth stack element below NEXT.

The DSP-relative offset refers to the position before the instruction executes.

See *Documenting Changes to the Stack* on page 40 for information about the importance of using stack comments like those in the example to document how the instructions affect the stacks.

BP Relative

In the BP-relative addressing mode, the instruction argument is a general-purpose register (R0 to R15). The register name is prefixed by an exclamation mark (!) to indicate a displacement relative to the base page.

Example:

```
pop  !R0            ; ( )
push !R2            ; ([R2], ...)
```

The **pop !R0** instruction pops the value of TOS into the R0 register (byte 8 of the base page). The **push !R2** instruction pushes the contents of the R2 register (byte 10 of the base page) into TOS.

BP Indexed

In the BP-indexed addressing mode, the instruction operand is TOS. The BP-indexed addressing mode uses the value in TOS as an offset into the base page, providing access to all 256 base-page bytes. The register name (TOS) is prefixed by an exclamation mark (!) to indicate a displacement relative to the base page.

Because this addressing mode uses TOS for its operand, instructions that use this addressing mode use NEXT for their argument.

Example:

```
pop  !tos           ; ( )
push !tos           ; ([TOS], ...)
```

The **pop !tos** instruction pops the value of NEXT from the stack and places it into the location within the base page specified by the value of TOS. The **push !tos** instruction pushes the contents of the location within the base page, specified by the value of TOS, into NEXT on the stack. That is, it pushes the TOS-th byte of the base page into NEXT.

Indirect

In the indirect addressing mode, the instruction operand is the value that is at the top of the return stack.

Example:

```
push  [rsp]         ; (z, ...)
```

The **push [rsp]** instruction pushes the contents of the top of the return stack (pointed to by RSP) into TOS on the data stack. RSP is not changed.

Another instruction that uses the indirect addressing mode is the **dbrnz** instruction, which uses this addressing mode for the branch destination. Its reference to [RSP] is implied.

Relative

In the relative addressing mode, the instruction takes a relative address for its operand; a distance rather than an absolute address (some instructions that use relative addressing take a second operand). The branch destination is computed by adding the relative address (the distance from the current IP) to the instruction pointer IP. Some instructions that use relative addressing support signed distances (that is, support branching in both directions), while other instructions support relative addressing for positive distances only (only jump forward).

Unlike absolute addresses, which always require a 16-bit value, relative addresses support shorter branch distances with the benefit of more compact machine code, which benefits both the memory footprint and execution time.

Example:

```
loop    ...
        brz loop  ;
```

The **brz** instruction branches to the relative address provided in its operand if the instruction's argument, the value in TOS, is zero. If TOS is not zero, this

conditional branch does not occur, and execution continues with the next instruction that follows **brz**.

3

Writing a Neuron Assembly Utility Function

This chapter describes general guidelines for writing a utility function in Neuron assembly language.

Overview of Stack-Oriented Programming

Neuron assembly language is a *stack-oriented* programming language, which means that the data stack is the primary tool for managing data manipulation, rather than using other tools, such as processor registers. The Neuron instruction set provides a rich set of stack-oriented instructions, rather than register-oriented instructions.

One of the major features of a stack-oriented language is its use of *postfix notation* (also known as Reverse Polish notation) for arithmetic and logical operations. This notation is well suited for working with a stack. Arithmetic and logical operations consist of numeric values and arithmetic or logical operators (for addition, subtraction, AND, OR, and so on). In postfix notation, the operators follow the operands.

To add three and four in postfix notation, you write "`3 4 +`"; "3" and "4" are the operands, and "+" is the operator. For more conventional infix notation, you add three and four by writing "`3 + 4`". For a stack-oriented language such as Neuron assembly language, you push the values "3" and "4" (the operands) onto the stack, and then use the **ADD** instruction (the operator) to add the two numbers together. The arithmetic operations consume the operand values on the stack and place the results of arithmetic operations onto the stack.

In postfix arithmetic, the affinity of operands to their operator is defined by the order in which the operands and operators appear. Parentheses are not required for postfix arithmetic.

Example: To perform the arithmetic operation that is conventionally written as "`3 – 4 + 5`" in postfix notation, you write "`3 4 – 5 +`". This operation is expressed in Neuron assembly language as:

```
PUSH #3
PUSH #4
SUB NEXT,TOS
PUSH #5
ADD
```

This code performs the following tasks:

1. The first two **PUSH** instructions place the values 3 and 4 onto the stack.
2. The **SUB NEXT,TOS** instruction consumes the values 3 and 4, and places the result of the subtraction (-1) on the top of the stack. The operands TOS and NEXT refer to the top of the stack and the next element below the top of the stack, respectively.
3. The next **PUSH** instruction places the value 5 on the stack so that the **ADD** instruction can add the two values (-1 and 5) together.
4. The **ADD** instruction consumes the values -1 and 5, and places the result of the addition (4) on the top of the stack. Note that because the addition operands are on the stack, the **ADD** instruction itself has no operands or parameters (that is, it uses implicit addressing).

The number signs or hashes for the **PUSH** instructions tell the assembler that the values are immediate, literal, values, rather than addresses (the instructions use immediate addressing). Because of the small data values, this example could have used the **PUSHS** instruction, rather than the **PUSH** instruction, to produce

more compact code; see *PUSHS (Push Short)* on page 110 for a description of the **PUSHS** instruction.

Note that the stacks and all data registers are 8 bits wide. Operations on larger scalars typically require the combination of more than one instruction. Chapter 8, *System-Provided Functions*, on page 159, describes system functions that can be used to accomplish many common tasks, including arithmetic with 16-bit operands.

Designing a Neuron Assembly Function

A general approach to writing Neuron Assembly functions is to push arguments for a called function onto the stack, call the function, push more arguments, call another function, and so on, and then finally, after the last operation, clean up the stack frame. This approach, however, can lead to suboptimal code and larger than necessary stack frames. For more efficient Neuron assembly programming, it often helps to look ahead into the remainder of the algorithm, and to get the stack layout correct as soon as possible.

Optimized assembly functions can often consume stack arguments rather than work with copies of the function arguments (which the caller would later dispose of).

Note that different rules might apply when interfacing Neuron C functions; see Chapter 4, *Interfacing with a Neuron C Application*, on page 47, for more information.

One way to create a correct stack layout that is suited for postfix arithmetic is to create a prefix formula, and then read that formula in reverse to generate a postfix formula that is appropriate for creating the stack frames. This technique generally applies to mathematical problems, but also applies to other types of problems, and is often easier for programmers who not used to postfix notation. Generate a prefix form of the algorithm, using function names for the operators. For example, use "add(A,B)" for the more conventional infix form "(A + B)". Then, read the expression from right-to-left to generate the postfix form "(B,A,add)". From the postfix form, you can generate the necessary stack frames: push A, push B, add.

Example:

For this extended example, assume that you need to write a function that computes a circular area, given a radius *R*. That is, the function has the following characteristics:

- Input: radius, R
- Output: area, πR^2

Assume that this function will be part of a mathematics library that you plan to write in Neuron assembly language.

The first decision to make is whether the function will be called from Neuron C applications or only from other Neuron assembly functions. For this example, the function will be called from Neuron C applications.

If you name the function "carea", its function prototype in C is:

```
// carea(radius) returns pi * radius^2
unsigned long carea(unsigned long radius);
```

The following tools are available to help you write this function:

- For the value pi (π), you can use the following approximation: 355/113 (3.1415929203…), which has an error relative to pi of $8.5 * 10^{-8}$.
- To multiply two unsigned long integers, you can use the **_mul16** system function.
- To calculate `(a*b)/c` with a 32-bit intermediate result, you can use the standard Neuron C **muldiv()** function:

```
unsigned long muldiv(unsigned long a,
                     unsigned long b,
                     unsigned long c);
```

- You can use general-purpose pointer registers for general 16-bit values, not just for pointer values.

Thus, one possible algorithm for the **carea** function is:

```
carea := π * r²
       = π * r * r
       = 355 * r * r / 113
       = (355 * r) * r / 113
```

which becomes:

```
carea = muldiv(355*r, r, 113)
```

which finally becomes:

```
carea = muldiv(_mul16(355, r), r, 113)
```

The proposed solution uses a fixed-point approximation for pi (355/133), and a combination of the **_mul16** and **muldiv()** library functions. To translate the solution into Neuron assembly instructions, read the algorithm from right to left:

1. Push the unsigned long value of 133 onto the stack.
2. Push the unsigned long radius value onto the stack. This value is the carea function's argument.
3. Push the unsigned long radius value onto the stack again.
4. Push the unsigned long constant 355 onto the stack.
5. Call the **_mul16** function.
6. Call the **muldiv()** function.

The Neuron assembly implementation of this function is shown below:

```
    IMPORT _mul16        ; (a(2), b(2) -- (a*b)(2))
    IMPORT muldiv        ; (a(2), b(2), c(2) -- (a*b/c)(2))

PIFACTOR   EQU d'355
PIDIVISOR  EQU d'113
    ; PIFACTOR/PIDIVISOR = PI = 3.14159…

SCRATCH16  EQU 0         ; 16-bit scratch register

%carea     APEXP         ; (r(2) -- (pi*r*r)(2))
    ; save r(2) to SCRATCH16 and put PIDIVISOR in place:
    popd   [SCRATCH16]   ; ()
    pushd  #PIDIVISOR    ; (piDiv(2))
```

```
; restore radius twice:
pushd  [SCRATCH16]   ; (r(2), piDiv(2))
pushd  [SCRATCH16]   ; (r(2), r(2), piDiv(2))

; push the "second half" of pi:
pushd  #PIFACTOR     ; (PiFac(2), r(2), r(2), PiDiv(2))

; do the math:
call   _mul16        ; ((piFac*r)(2), r(2), piDiv(2))
brf    muldiv        ; ( -- carea)
```

For this implementation, the Neuron assembly file imports the two external functions: **_mul16** and **muldiv**. The **_mul16** function has a preceding underscore because it is intended for use only with assembly functions. Note that neither the **_mul16** nor the **muldiv** function names has a percent sign (**%**) prefix. The **_mul16** system function is not defined in, or designed for, the Neuron C application name space. The **muldiv** function is designed for the Neuron C application name space, but is imported using the **system** keyword in Neuron C, which makes the symbol available without the percent sign (**%**) prefix.

The assembly file then defines two mnemonics for the pi value approximation, and defines a mnemonic for one of the general-purpose 16-bit registers. This register temporarily holds a 16-bit value; as long as we do not interpret such a value as a pointer, these registers can hold any 16-bit value, even though they are typically used for pointer values.

The **%carea** function is then declared, using the percent sign (**%**) prefix, with the **APEXP** directive to make this symbol available to the Neuron C application name space.

The **%carea** function then sets up the stack for calling the **_mul16** and **muldiv** functions. Note that because the **_mul16** function is located in the near code area, it can be called with the two-byte **CALL** instruction, rather than the slightly less efficient three-byte **CALLF** instruction.

By setting the stack up correctly, the **carea** function can branch into the **muldiv** function (using the **BRF** instruction), rather than having to call the **muldiv** function (using the **CALLF** and **RET** instructions). The **carea** function can use the branch instruction because the stack layout for executing the **muldiv** function is correct, and contains only the arguments for the **muldiv** function. Also, the result type of the **carea** function is the same as that of the **muldiv** function.

For more information about the **_mul16** function, see *_mul16 (Multiply, 16 Bit)* on page 183. For more information about the **muldiv()** function, see the *Neuron C Reference Guide*.

Interrupt Tasks with Assembly Code

For Series 5000 devices, you can write an interrupt task to handle hardware interrupts. Although you write the interrupt task in Neuron C, you can include a Neuron assembly function in the Neuron C task using the **#pragma include_assembly_file** compiler directive, or call a function written in assembly from your interrupt task.

In general, interrupts run in a separate processor context on the Series 5000 Chip, but for the lowest two clock rates (clock multiplier values ½ and 1 in the

device's hardware template), the interrupts run on the application processor, and share stacks and common registers with the Neuron C application.

Recommendation: To ensure that shared and global data is updated correctly by both the application and an interrupt, use the locking construct (**__lock** keyword in Neuron C; **io_iaccess()** and **io_irelease()** functions in Neuron assembly) to provide semaphore access to shared and global data.

See the *Neuron C Programmer's Guide* for more information about interrupts and the **__lock** keyword. See *io_iaccess (Acquire Semaphore)* on page 170 and *io_irelease (Release Semaphore)* on page 171 for information about these functions.

Documenting Changes to the Stack

Commenting source code can greatly improve the readability and maintainability of the code. The time that you spend writing a useful comment will be repaid when you need to maintain or debug the code.

In addition to plain-language comments that describe the purpose, assumptions, and side effects of the code, Neuron assembly language code can greatly benefit from comments that show how the stack changes. Including comments that describe stack changes can help you to:

- Improve the readability of the code by showing its expected behavior
- Find potential stack errors, such as unbalanced stacks or data that is on the wrong stack
- Determine offset values for DSP-relative addressing

Although the Neuron Assembler does not enforce a particular commenting style for stack-related comments, this section describes a recommended comment style. This recommended style is also used throughout this book in the code examples.

There are two types of stack-related comments:

- Stack-effect comments
- Stack-transformation comments

Stack-effect comments describe how a particular instruction or system-provided function call alters the stack. For example, the **PUSH** instruction adds a value to the top of the stack, and thus alters the stack.

Stack-transformation comments describe how an entire assembly function alters the stack. These comments show the expected contents of the stack before and after the function runs. For example, before a function runs, the stack contains the function arguments, and after it runs, the stack contains the function result or return code.

Both types of stack-related comments use parentheses to denote the boundary of the stack. The data stack is shown first, and if the return stack is affected, it is shown second with an *R* prefix. For example, the following comment shows a change to both stacks:

```
; (cs, pData) R(size)
```

The example shows two elements on the data stack, and one element on the return stack. The following sections describe how to read these comments.

Stack-Effect Comments

In general, the data stack is documented as a set of ordered values from left to right, with TOS in the first (left-most) position. For example, if the values 1 and 2 were on the data stack, with 1 in TOS and 2 in NEXT, and the current instruction is a **PUSHS** instruction to push the value 0 onto the stack, the stack-effect comment would be `(0,1,2)`, as shown below:

```
pushs #0            ; (0, 1, 2)
```

The comment follows the instruction, so that the stack-effect comment shows the stack layout after the instruction has been executed. In this case, the comment shows that 0 has been pushed onto the stack, and the previous values 1 and 2 have been moved to the right (down the stack).

Note that this stack-effect comment does not show the entire stack, that is, the data stack contains more than the three values 0, 1, and 2. Instead, the comment shows only enough stack elements to clarify how the stack is changed by the current instruction or function call. You could add an ellipsis (`...`) or a double hyphen (`--`) to the right of the changed values to remind you that there is more data on the stack (for example, `(0,1,2,…)`). However, the ellipsis or double dash is not necessary.

In general, you do not know the actual data values on the stack, but instead you are pushing and popping variables. Add the variable name (or a portion of it) to the stack comment, for example:

```
push interface_type      ; (type)
```

Stack values are unsigned one-byte scalars unless stated otherwise.

Multi-Byte Values

For multi-byte values, add the element's size, in bytes, to the value in the comment. Enclose the size in parentheses following the value itself. For example, for a 16-bit value named *address*, the comment would read as `address(2)`. Therefore, the following two stack-effect comments are equivalent:

```
; (address(2), …)
; (addresslo, addresshi, …)
```

Pointer Values

For pointer values, use square brackets (`[]`) to indicate that the value is fetched from an address. For example, the following stack-effect comment describes one byte from the location *address(2)*:

```
; ([address(2)])
```

Conditional Values

Occasionally, stack elements are conditional. For example, the **BRNEQ** instruction drops TOS if the comparison (which is part of the **BRNEQ** operation) results in equality, but leaves TOS intact otherwise. Use braces (`{}`) to indicate conditional stack elements. For example, the following comment indicates that

the *code* element might still be on the stack (if the equality test fails), or might no longer be on the stack (if the equality test succeeds):

```
brneq #d'123,next_test   ; ({code})
```

Showing the Return Stack

Because most instructions and functions only affect the data stack, stack comments refer to the data stack by default. For those cases where data elements are handled on both the data and return stacks, show the effect to both stacks in the comment. Use the *R* prefix to denote the effect to the return stack. The following example shows the data stack containing element *a*, and the return stack containing data element *b* following the **POPPUSH** instruction:

```
poppush              ; (a) R(b)
```

Recommendation: Show return stack comments only where necessary. When the return stack is no longer affected, the first instruction that no longer uses the return stack should show an empty return stack as "R()". In this way, the code shows that the return stack is empty of local variables and is not needed, and shows that the return stack is not accidentally forgotten.

The following code saves a copy of the value in TOS on the return stack temporarily, and restores it after a function call:

```
push tos             ; (z, z)
poppush              ; (z) R(z)
callf myFunction     ; ( ) R(z)
pushpop              ; (z) R()
```

Stack-Transformation Comments

A stack-transformation comment does not apply to an individual operation, but to an entire function, and typically follows the **EXPORT** or **APEXP** directive. The stack transformation comment shows the expected stack layout at the beginning of the function (the function arguments), a double hyphen (as a separator), and the data stack elements of relevance after the function completes.

For example, the stack transformation comment for the assembly version of a **strlen()** equivalent function, which takes a pointer argument and returns an unsigned integer result, would be:

```
  %strlen  APEXP              ; (pString(2) -- length)
```

Thus, `pString(2)` represents the 2-byte function argument, and `length` represents the function return value.

You could think of stack transformation comments as the Neuron Assembly equivalent of Neuron C function prototypes. However, the Neuron Assembler does not verify the "prototype" against the actual implementation.

Integrating the Program

The most common method of integrating Neuron assembly language functions in a Neuron C program is to use the **#pragma include_assembly_file** compiler directive. During compilation of the Neuron C program, this directive causes the

Neuron C compiler to copy the contents of a Neuron assembly file as-is into the assembly-language output for the program, as shown in **Figure 8**.

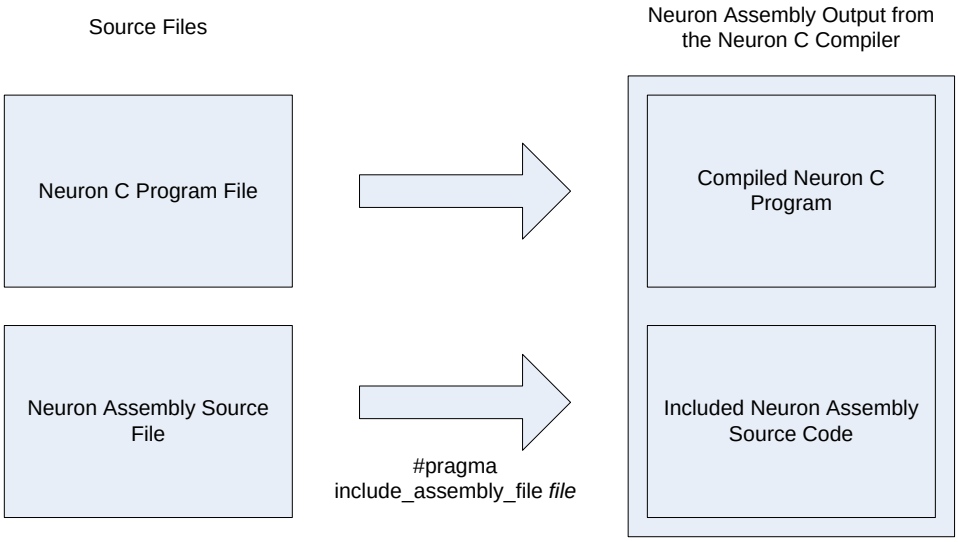

Figure 8. Including Assembly Code in a Neuron C Program

This method exposes the Neuron C application programmer to your assembly source code. For improved code management, and the option to conceal your assembly source code from the Neuron C programmer, see *Assembling the Program*.

For many general cases where a Neuron C application is supported by one or a few functions written in Neuron assembly, this method is preferred because of its simplicity. However, you should consider the following limitations:

- If you include assembly source within a function, interrupt task, or when task, the CODE segment becomes active, and you cannot change the segment.
- If you include assembly source outside a function, interrupt task, or when task, you must specify the segment to use, because you cannot make assumptions about the current segment type.
- The compiler uses **RADIX HEX**, and thus assumes the default format for numerical constants is hexadecimal. Do not change the default radix (or be sure to change it back to hexadecimal) when including assembly source files.

The **#pragma include_assembly_file** compiler directive is available for users of either the NodeBuilder Development Tool or the Mini EVK.

Assembling the Program

To assemble a single Neuron assembly file, use the Neuron Assembler command-line tool (**NAS.EXE**).

Example: To assemble a file named **checksum.ns**, use the **NAS** command, as shown below:

```
C:\myDevice> nas --listing checksum.ns
Neuron Assembler, version 5.00.21, build 0
```

```
Copyright (c) Echelon Corporation 1992-2009

0 warning(s), 0 error(s)
```

The `--listing` option causes the Assembler to generate a listing file. The Neuron Assembler creates two output files:

- **checksum.nl** (the listing file)
- **checksum.no** (the object file)

To add an object file to a Neuron library, use the Neuron Librarian command-line tool (**NLIB.EXE**).

Example: To add the previously assembled **checksum.no** file to an existing library named **wwdc.lib**, use the **NLIB** command as shown below:

```
C:\myDevice> nlib --add wwdc.lib checksum.no
Neuron C Librarian, version 5.00.21, build 0
Copyright (c) Echelon Corporation 1992-2009

0 warning(s), 0 error(s)
```

You can add multiple files to a library at once, or you can add each file separately by running the **NLIB** command several times. You can use the `--report` option to inspect an existing library and list its contents.

See *Neuron C Compiler for Assembly Programming* on page 3 for more information about compiling a Neuron C program that includes Neuron assembly code. See *Neuron Assembler Command Line Tool* on page 3 for more information about the NAS command. See the *Neuron C Programmer's Guide* for more information about the Neuron Librarian.

Linking the Program

To link an assembled object file with a compiled Neuron C object file, refer to your Neuron C development tool's user's guide. Both the NodeBuilder FX Development Tool and the Mini FX Evaluation Kit support user-defined libraries, and a simple mechanism to specify them.

Debugging the Program

Although the NodeBuilder FX Development Tool allows you to debug Neuron C code, it does not provide support for debugging Neuron assembly code directly. That is, you cannot set a breakpoint in assembly code, single-step through assembly code, or watch variable or register values.

Thus, to debug a Neuron assembly function, you should create helper functions in Neuron C to call and test the assembly function. For example, for a **Checksum()** function written in Neuron assembly language, you should create a test function that calls the assembly function and verifies its correctness, as shown below:

```
#pragma include_assembly_file "checksum.ns"
extern unsigned Checksum(unsigned, const char*);

void TestCall(void) {
  unsigned cs;
  cs = Checksum(8, "ABCDEFGH");
```

```
}
```

In addition, you can add a callback function to the test application, and modify the assembly source to call the Neuron C callback function. This callback can perform diagnostic functions, or you can use it simply to set a breakpoint in the NodeBuilder Debugger. The **diagnosis()** function, shown below, is a callback that serves as a breakpoint for the debugger:

```
#pragma include_assembly_file "checksum.ns"
extern unsigned Checksum(unsigned, const char*);

void TestCall(void) {
  unsigned cs;
  cs = Checksum(8, "ABCDEFGH");
}

// set breakpoint here
void diagnosis(unsigned currentChecksum,
               unsigned remainingSize,
               const char* remainingData) {
}
#pragma ignore_notused currentChecksum
#pragma ignore_notused remainingSize
#pragma ignore_notused remainingData
#pragma ignore_notused diagnosis
```

The **#pragma ignore_notused** directives allow the compiler to ignore the warnings that it otherwise generates for unused variables and functions. The compiler would generate warnings for these variables and the function because the compiler is not aware of the embedded assembly code, because the assembly code is already compiled.

See *Calling a Neuron C Function from Assembly Code* on page 52 for a description of this **diagnosis()** function.

Another debugging technique is to use dedicated global variables that report status details which are internal to the assembly function. Because they are global variables, you can access them from your Neuron C program. This technique is especially useful for assembly functions that do not complete at once. For example, a state machine typically runs repeatedly until it returns to an idle state.

It is also strongly recommended that you follow strict and thorough stack commenting policies (a recommended policy is described in *Documenting Changes to the Stack* on page 40). Although thorough stack commenting does not help in debugging a problem, it is a principal tool to help avoid the need for debugging.

4

Interfacing with a Neuron C Application

This chapter describes the conventions and interface requirements for a Neuron assembly function to work with a Neuron C application program.

Overview

Typically, you use Neuron assembly language to create utilities that can be used with an application that is written in Neuron C. The Neuron C program code might call functions defined in Neuron assembly, or a Neuron assembly function might call a Neuron C function or program.

This chapter describes the conventions that a Neuron assembly language function must follow when interfacing with Neuron C functions, including data, calling and naming conventions.

See the following sections for information about integrating, assembling, and linking your assembly program with the Neuron C application: *Integrating the Program* on page 42, *Assembling the Program* on page 43, and *Linking the Program* on page 44.

Naming Conventions

Within a Neuron assembly file, declare functions, variables, or any other linkable item that interfaces with the Neuron C application, with a percent-sign prefix. This declaration allows the symbol to become part of the Neuron C name space.

To make the symbol and its meaning available to the Neuron C application, you must provide its C language 'extern' specification (for variables and constants) or its prototype (for functions).

Within a Neuron C file, when you specify an external symbol, the Neuron C compiler automatically assumes a percent-sign prefix.

Example:

The following example includes an assembly-coded function in a Neuron C program:

```
#pragma include_assembly_file "checksum.ns"
extern unsigned Checksum(unsigned, const char*);
```

The compiler-generated calls to this function from the Neuron C application use the name `%Checksum`. Thus, the assembly function must use the `%Checksum` label with the **APEXP** directive:

```
%Checksum  APEXP
           ...                  ; function body
           ret                  ; return to caller
```

The Neuron C Compiler (and its companion tools, such as the Neuron Assembler or Linker) cannot verify that the prototype (or extern specification that you provide) actually matches the implementation in the assembly source file. It is the programmer's responsibility to ensure that the specification in Neuron C matches the implementation in Neuron Assembly.

Function Parameters

Typically, an assembly language function is called with parameters. The compiler pushes function arguments on the stack from right-to-left, so that the last argument is in the TOS element of the stack. For example, for a function

with two arguments, such as *Checksum(unsigned, const char*)*, the *const char** pointer is pushed first, and the *unsigned* argument is pushed into TOS.

In typical C-language implementations of such functions, the caller is responsible for removing the arguments from the stack after the function completes. But for Neuron assembly language functions, the called function must ensure that all arguments are consumed (removed from the stack) before returning to the caller. Only function results can remain on the data stack. This calling convention typically allows for more compact code, given the Neuron Chip's stack-oriented architecture.

Calling Conventions

Depending on its use, an assembly language function can use different calling conventions:

- Assembly-coded functions that are used exclusively by other assembly-coded functions can employ any appropriate calling conventions for efficiency or ease of use. For example, a function could pass pointer arguments through general-purpose pointer registers.
- Any function that calls, or might be called from, code that is generated by the Neuron C compiler must meet the guidelines described in this section.

To avoid naming collisions, the Neuron C compiler enforces the following naming convention for global and static data:

- Firmware labels begin with alphabetic characters.
- Firmware labels that are used as interfaces for application code are prefixed with the underscore (_) character.
- Global labels are prefixed with the percent sign (**%**). For a Neuron C variable or function name, the **%** character is added by the compiler, not the programmer. For Neuron assembler functions or data that interface with Neuron C, you must be sure to add the **%** character for all exported labels. When importing symbols from a Neuron C application into your assembly programming project, you must also add the **%** character for all imported symbols.
- Static labels are also prefixed with the percent sign (**%**). A Neuron C static segment label is the **%** character plus name of the module (without the file extension) plus another **%** character and a unique number generated by the compiler. For example, an anonymous string "ABC" argument for a function call within a file named **MyDevice.nc** might be named %MYDEVICE%STR1 by the compiler.

Your Neuron assembly functions must follow the same naming convention so that the functions can interface with Neuron C programs.

Functions exchange arguments and results on the data stack. When calling a function, the caller pushes the function's arguments onto the data stack, right to left, in the order of the Neuron C function prototype.

However, unlike most implementations of the C language, in Neuron C, the *called function*, not the caller, is responsible for clearing the arguments off the data stack prior to returning to the caller. Thus, the called function must leave only function results (if any) on the data stack.

This optimization is leads to more efficient code (as the called function might consume its arguments rather than working on copies), and leads to more stack-efficient programs.

Example:

This example assumes a utility written in Neuron assembly that has the following function prototype in Neuron C:

```
extern unsigned Checksum(unsigned size, const char* pData);
```

When calling this function, the pointer argument *pData* is pushed onto the stack first (right to left), followed by the *size* argument. When the function call returns, the unsigned result of the Checksum function is found on top of the data stack. The *pData* and *size* arguments are no longer available on the data stack.

Consider the following Neuron C code, which calls the assembly-language code function *%Checksum* in the **Testcall** function:

```
#pragma include_assembly_file "checksum.ns"
extern unsigned Checksum(unsigned, const char*);

void Testcall(void) {
  unsigned cs;
  cs = Checksum(8, "ABCDEFGH");
}
```

For this Neuron C code, the Neuron C compiler translates the **Testcall** function into the following assembly code (taken from the assembly listing of the compiler-generated code):

```
                    IMPORT      %Checksum

                    SEG   CODE
                    ORG
0000              _%MYDEVICE%STR1     EXPORT
0000  41  42        DATA.B      41,42,43,44,45,46,47,48
0002  43  44  45  46
0006  47  48
0008  00            DATA.B      0

                    SEG   CODE
                    ORG
0000              %Testcall     APEXP ; Function body
0000  C0            ALLOC #1
0001  B5 0000*      PUSHD #_%MYDEVICE%STR1
0004  B4 08         PUSH  #8
0006  77 0000*      CALLF %Checksum
0009  E5            DROP  NEXT
000A  7F            DEALLOC     #1
```

The listed code performs the following tasks:

1. First, the compiler generates an **IMPORT %Checksum** statement as result of the Neuron C function prototype specification.
2. In the first code segment (**SEG CODE**), the compiler allocates initialized data for the anonymous constant string variable using the compiler-generated name `_%MYDEVICE%STR1` (the "MYDEVICE" part of the name is derived from the name of the Neuron C file). The **DATA.B** directive

also supports a string format, but does not automatically include the terminating zero-byte. An alternative declaration would have been `_%MYDEVICE%STR1 DATA.B "ABCDEFGH", 0`.

3. In the second code segment (**SEG CODE**), the **Testcall()** function (**%Testcall APEXP**), uses the **ALLOC #1** instruction to allocate one local variable on the data stack (for the unsigned *cs* local variable). It then pushes the address of the string onto the stack using the immediate push-double instruction **PUSHD #_%MYDEVICE%STR1**, followed by a **PUSH #8** instruction for the size parameter. Note that the value 8 is out of range for the shorter **PUSHS** instruction (which has a 1-byte opcode), so the **PUSH** instruction must be used instead (which has a 2-byte opcode).
4. Then, the Checksum function is called with the **CALLF** instruction, a far call that can target any location in the 64 KB address space.
5. Finally, the compiler assigns the function result (in TOS) to the local variable (in NEXT) with a simple **DROP NEXT** instruction. The *cs* local variable is now in TOS. With the end of the block (indicated by the closing curly brace), the compiler-generated code frees the allocated stack space with the **DEALLOC #1** instruction. The **DEALLOC** instruction also performs a return-from-call operation (equivalent to **RET**).

Data Representation

The following sections describe how various types of data are represented in the Neuron assembly language. The data types described include integers, characters, strings, multi-byte objects, arrays, structures, unions, and bitfields.

Integers, Characters, and Strings

Short integers and characters are stored as a single byte. The Neuron assembly representation of a signed integer is in two's-complement form; thus, the same instructions can be used for both signed and unsigned arithmetic. A variety of firmware functions are used for other arithmetic operations, as described in Chapter 8, *System-Provided Functions*, on page 159.

A character string is an array of characters (thus, an array of bytes). Strings that interface with Neuron C are expected to include a terminator byte of zero.

Each Neuron C global or static variable is defined as its own relocatable segment within the appropriate segment type.

Multi-Byte Values

Multi-byte scalar objects (such as signed and unsigned long integers) are stored as two consecutive bytes in big endian orientation: the most significant byte is stored at the lower address. Whenever a long integer is pushed onto the data stack, the most significant byte is pushed first (or popped last).

Arrays

The Neuron Chip and Smart Transceiver hardware has no data alignment requirement, beyond the natural byte alignment. Thus, arrays are consecutive elements without intervening padding. The first element (the element at index 0) is at the lowest address of the array. The remaining elements appear consecutively at addresses that increase as multiples of the array-element size.

Arrays are therefore packed objects with no unused space between the elements. Array sizes are always exactly the size of an individual element multiplied by the number of elements in the array.

Structures and Unions

A structure is a group of fields packed into consecutive byte locations. The first field is at the lowest address of the structure. The remaining fields appear consecutively at addresses that increase by adding the size of the previous fields to the beginning address of the structure. The size of a structure is always exactly the sum of the sizes of each individual element in the structure.

A union is a group of overlaid fields. Each field starts at the beginning of the union. The size of a union is always exactly the maximum of the sizes of each individual field in the union.

Bitfields

Bitfields in a Neuron C structure or union are always packed within one or more bytes. A single bitfield cannot cross a byte boundary; thus, the maximum bitfield size is eight bits. The first bitfield in a byte uses the most significant bits, and subsequent bitfields within the byte use the next most significant bits until all bits are used.

Anonymous bitfields of non-zero size are supported as placeholders, allowing for exact positioning of the named bitfields within their container byte. Anonymous bitfields of size zero always complete the current container.

A new byte is begun when a bitfield is encountered that is too large for the remaining bits, or when an anonymous bitfield is encountered with size 0, or when a non-bitfield object is encountered.

Calling a Neuron C Function from Assembly Code

When a Neuron assembly function calls a Neuron C function, the main task of the assembly function is to prepare the stack for the function by pushing the function arguments in the correct order, and to call the function. Because the called function clears the stack as part of its operation, the calling assembly function need only retrieve the function's result data (if any) from the stack.

The general approach to calling Neuron C functions from assembly code includes performing the following steps:

1. Import the symbol for the function.
2. Preserve all general-purpose registers in use.
3. Push the function arguments onto the data stack, from right to left.

4. Call the function.
5. Process optional function results and clear the stack frame.
6. Restore general-purpose registers saved.

Example:

The following example shows how to set up and call a diagnosis function that is defined in Neuron C.

The Neuron C definition for this function is:

```
void diagnosis(unsigned currentValue,
               unsigned remainingSize,
               const char* remainingData){
...
}
```

The following Neuron assembly language code shows how to call this Neuron C function:

```
pData    EQU  0

IFDEF    _DEBUG
IMPORT   %diagnosis          ; (cs, size, pData -- )
         ...                 ; (cs) R(size)
         pushd [pdata]       ; (pData(2), cs) R(size)
         pushd [pData]       ; (pData(2), pData(2), cs) R(size)
         push  [rsp]         ; (size, pData(2), pData(2), cs) R(size)
         push  [dsp][-4]     ; (cs, size, pData(2), pData(2), cs) R(size)
         callf %diagnosis    ; (pData(2), cs) R(size)
         popd  [pData]       ; (cs) R(size)
ENDIF
```

Thus, the Neuron assembly code performs the general steps for calling the Neuron C function:

1. Import the diagnosis symbol (**IMPORT %diagnosis**).
2. Preserve all general-purpose registers in use (**pushd [pData]**).
3. Push the function arguments onto the data stack, from right to left (**pushd [pData]**, **push [rsp]**, **push [dsp][-4]**).
4. Call the diagnosis function (**callf %diagnosis**).
5. Process optional function results and clear the stack frame. This example does not show processing for function results.
6. Restore general-purpose registers saved (**popd [pData]**).

This implementation uses conditional assembly based on a predefined symbol **_DEBUG**. Using conditional assembly allows this function to be called only within debug builds. However, conditional assembly is not available with Neuron C programs; you must create a Neuron assembly code library and link the library with the Neuron C program.

The assembly function pushes the pointer register P0 on to the stack twice (**pushd [pData]**): once to save the register content across the call to the diagnosis function, and a second time to provide the current pointer value to the diagnosis function itself. An alternative implementation could push the pointer once (**pushd [pData]**), then duplicate the value on the stack using two **push next**

instructions, but this alternative implementation would use one more byte of code.

In this example, it is assumed that the value for the *remainingSize* argument resides on top of the return stack. The function fetches a copy of it with the non-modifying **push [rsp]** instruction.

At this point, the value for the *currentValue* argument is now buried deep under the set of arguments and the copies of the pointer register. The function uses DSP-relative addressing to access this value. The offset for retrieving the value is -4, that is, the *currentValue* variable is the 4th byte on the data stack following NEXT (note that pData is a two-byte value). Thus, the function uses the **push [dsp][-4]** instruction.

Important: When using DSP-relative addressing, you should frequently review its use. As you modify the assembly implementation, the stack frames could change, which could require you to recalculate the offset values used for DSP-relative addressing. The Neuron Assembler provides no tools for to automate this recalculation. You can declare symbols (with the **EQU** directive) to assist with DSP-relative offsets, which can allow you to review symbol definitions rather than specific offsets, but the process is nonetheless manual.

The assembly function finally calls the diagnosis function with a **callf %diagnosis** function call, which pushes the instruction pointer (IP) onto the return stack, and executes the diagnosis function. If you knew that the diagnosis function would be located in nearby memory, this function call could have perhaps been replaced by the more compact **callr %diagnosis** instruction, which covers function calls in the -128..+127 distance range. However, you do not have control over the placement of the Neuron C-coded function, and thus must use the **CALLF** instruction.

Finally, when function processing is complete, the calling function restores the previously preserved register with a **popd [pData]** instruction.

5

Exploring an Example Function in Neuron Assembly

This chapter describes an example function written in Neuron assembly language. This example function is designed to be called by a Neuron C application program.

Overview of the Checksum Example

The checksum example function described in this chapter demonstrates the process of writing and testing a function in Neuron assembly language.

The example function calculates a simple checksum for a data string of a specified length. The checksum algorithm used combines all bytes with an exclusive OR. An equivalent Neuron C function for the checksum example is shown below:

```
unsigned Checksum(unsigned size, const char* pData) {
  unsigned result;
  result = 0;
  while (size--) {
    result ^= *pData++;
  }
  return result;
}
```

Although there might not be a need to recode this particular function in assembly language, this example demonstrates how to pass arguments from a Neuron C program to a Neuron Assembler function, how to manage the stacks to process a simple calculation, and how to return arguments from a Neuron Assembler function to a Neuron C program.

Implementing the Checksum Function

A Neuron assembly implementation of the checksum example function is shown below:

```
            SEG CODE
            ORG

pData       EQU 0

%Checksum   APEXP             ; (size, pData(2) -- cs)
            poppush           ; (pData(2)) R(size)
            popd  [pData]     ; () R(size)
            pushs #0          ; (cs) R(size)
cs_loop
            push  [pData][0]  ; (v, cs) R(size)
            inc   [pData]     ; (v, cs) R(size)
            xor               ; (cs) R(size)
            dbrnz cs_loop     ; (cs) R({size})
            ret               ; (cs) R()
```

The Neuron assembly code for the Checksum function performs the following tasks:

- The **EQU** directive defines a mnemonic (pData) with value 0 (zero). This mnemonic is used to identify one of the general-purpose pointer registers (P0). This example uses a mnemonic name (pData) that is more meaningful to this source code than the register name (P0).
- The **POPPUSH** instruction takes the value in TOS (the *size* argument) from the data stack and pushes it on the return stack. TOS now has the

value formerly held in NEXT. Using this instruction prepares for the loop construct that follows.

- The **POPD** instruction takes two bytes from the stack (the *pData* variable) and loads them into the general purpose register (P0, which was equated to the mnemonic name *pData*).
- The **PUSHS** instruction pushes a short immediate constant (value range 0..7) onto the stack. In this case, it pushes the value zero. The single **PUSHS** instruction allocates the local *cs* variable, and initializes it to the value zero.
- The **PUSH [pData][0]** instruction uses the indirect-indexed addressing mode to push the first byte in the array (index 0) addressed through the *pData* value (pointer register P0) onto the stack.
- The **INC [pData]** instruction uses pointer-direct addressing to increment the *pData* pointer (P0).
- The **XOR** instruction combines TOS and NEXT with a logical exclusive-OR, leaving the result in TOS. Prior to this operation, TOS contains the current byte read from the *pData* array, and NEXT contains the running local *cs* variable.
- The **DBRNZ** instruction completes the loop. The instruction first decrements the value on the top of the return stack, which holds the remaining size of the array that is being processed. If the resulting value is zero, the loop index is removed from the return stack and processing continues with the next instruction. If the resulting value is not zero, however, processing continues at the indicated label *cs_loop*.
- The **RET** instruction returns to the caller function by loading the instruction pointer (IP) with the return address from the return stack (which was below the loop counter). This return address is automatically loaded onto the return stack when the Neuron C program calls the function with the **CALL**, **CALLR**, or **CALLF** instruction.

Both stacks must be balanced when the function terminates: the data stack must hold the correct number of bytes to match the return type at every return point from the function, and the return stack should have the same number of bytes as when the function was called. That is, unlike in many common C language implementations, Neuron C and Neuron Assembly require that the called function itself remove the function arguments from the data stack.

Including the Assembly Function in a Neuron C Application

To include the **Checksum** function in a Neuron C application program, use the **#pragma include_assembly_file** compiler directive. This directive instructs the compiler to copy the content of the specified file directly into the assembly output for the Neuron C program. This directive is available to both Mini FX and NodeBuilder users, and is the preferred method for working with simple assembly language functions.

The following Neuron C program includes a **TestCall()** function that calls the assembly-coded Checksum function:

```
#pragma num_alias_table_entries 0

#pragma include_assembly_file "checksum.ns"
extern unsigned Checksum(unsigned, const char*);

void TestCall(void) {
  unsigned cs;
  cs = Checksum(8, "ABCDEFGH");
}
```

The Neuron C compiler reads the **Checksum** function arguments from right to left, and pushes them onto the stack so that the first argument (*unsigned*) is in TOS, the top of the stack. When the function completes, the compiler expects the function result (*cs*) on the data stack. Thus, the function must digest all arguments and local variables before returning to the caller.

The **#pragma include_assembly_file** compiler directive instructs the compiler to include the assembly source file, but the compiler does not parse the included assembly file. Likewise, the compiler does not analyze the included assembly code to ensure that there is an appropriate Neuron C function prototype for the assembly implementation. You must provide a proper function prototype, typically in a header file. In addition, you muse ensure that the number, order, and type of arguments, and any results, are correct, both in the Neuron C function prototype and in the Neuron assembly implementation.

Note that certain Neuron C specifications have no direct correspondence in a Neuron assembly function. For example, the declaration of the *pData* argument as **const** (a variable pointer to constant data) is correct from a C language point of view, but it is your responsibility in the assembly function to ensure that this data is not modified.

6

Neuron Assembly Language Instruction Statements

This chapter describes the Neuron assembly language instruction statements that are supported by the Neuron Assembler.

Overview of the Assembly Language Instructions

The Neuron assembly language provides instruction statements that control the functionality of the function or program. This chapter describes the Neuron assembly instructions that are supported by the Neuron Assembler.

Table 9 lists the Neuron assembly language instructions, grouped by general function: arithmetic operations, logical operations, shift and rotate operations, stack manipulation, conditional program control operations, unconditional program control operations, and other operations. Some instructions appear in more than one group because they perform more than one function.

Table 9. Neuron Assembly Instructions

Arithmetic Operations	
ADC	Add with carry
ADD	Add
ADD_R	Add and return
DBRNZ	Decrement and branch if not zero
DEC	Decrement
DIV	Divide
INC	Increment
MUL	Multiply
SBC	Subtract with carry
SUB	Subtract
Logical Operations	
AND	And
AND_R	And and return
NOT	Not
OR	Or
OR_R	Or and return
XOR	Exclusive or
XOR_R	Exclusive or and return

Shift and Rotate Operations	
ROLC	Rotate left through carry
RORC	Rotate right through carry
SHL	Shift left
SHLA	Shift left arithmetically
SHR	Shift right
SHRA	Shift right arithmetically
Stack Manipulation	
ALLOC	Allocate
DEALLOC	Deallocate and return
DROP	Drop from stack
DROP_R	Drop from stack and return
POP	Pop from stack
POPD	Pop pointer direct
POPPUSH	Pop from data stack and push onto return stack
PUSH	Push onto stack
PUSHD	Push pointer direct
PUSHPOP	Pop from return stack and push onto data stack
PUSHS	Push short
XCH	Exchange
Conditional Program Control Operations	
BRC	Branch if carry
BRNC	Branch if not carry
BRNEQ	Branch if not equal
BRNZ	Branch if not zero

BRZ	Branch if zero
DBRNZ	Decrement and branch if not zero
SBRNZ	Short branch if not zero
SBRZ	Short branch if zero
Unconditional Program Control Operations	
ADD_R	Add and return
AND_R	And and return
BR	Branch
BRF	Branch far
CALL	Call near
CALLF	Call far
CALLR	Call relative
DEALLOC	Deallocate and return
DROP_R	Drop from stack and return
OR_R	Or and return
RET	Return from call
SBR	Short branch
XOR_R	Exclusive or and return
Other Operations	
NOP	No operation

The following sections describe each of the Neuron Assembler instructions. The description for each instruction includes a table with the following information:

- The hexadecimal representation of the instruction's opcode
  The opcode is the part of a machine instruction that informs the processor of the type of operation to be performed and the type of data to be used. The first byte of an instruction is its opcode, and remaining bytes are its data. Some Neuron machine instructions have variable opcodes because data is included in the opcode so that the instruction's size can be as small as possible.
- The size of the instruction, in bytes
  Each machine instruction requires either one, two, or three bytes of

program memory. The size includes both the opcode and the instruction's data, if any.

- The number of CPU cycles required to execute the instruction
  Each machine instruction requires a number of processor clock cycles (within the Neuron Chip or Smart Transceiver) to execute. The number of cycles ranges from one to seven, depending on the type of operation.
- The instruction's affect on the Carry flag
  Many machine instructions have an affect on the Carry flag, and some use the contents of the Carry flag to perform their operation. The Carry flag can be used, modified, or cleared by an instruction.

The description for each instruction also describes whether the instruction applies to Series 3100 devices, Series 5000 devices, or to both device series. Most instructions apply to both series.

ADC (Add with Carry)

The **ADC** instruction adds two numbers with the value of the Carry flag. The **ADC** instruction uses the implicit addressing mode. The **ADC** instruction retrieves both TOS and NEXT from the data stack and adds them together with the value of the Carry flag. TOS and NEXT are consumed, and the result is placed in TOS. The operation modifies the Carry flag as result of the unsigned addition.

The **ADC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **ADC** instruction requires no operands:

```
ADC
```

Table 10 describes the attributes of the **ADC** instruction.

Table 10. ADC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
ADC	54	1	4	Used, modified

Example:

The following example performs the operation 2+3. The example assumes that the Carry flag is set.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
adc                ; (6, -, -)
```

The value of TOS after this code executes is 6 because 2+3+Carry = 6. In this case, the value of the Carry flag is used for the addition, and then modified (cleared) because the unsigned addition did not require an extra carry bit.

ADD (Add)

The **ADD** instruction adds two numbers. The **ADD** instruction uses one of two addressing modes:

- In implicit addressing mode, the **ADD** instruction retrieves both TOS and NEXT from the data stack and adds them together. TOS and NEXT are consumed, and the result is placed in TOS. The operation modifies the Carry flag as result of the unsigned addition.
- In immediate addressing mode, the **ADD** instruction (**ADD #*number***) adds a specified one-byte constant *number* to the value of TOS, and the result is placed in TOS. The value of *number* must resolve at link time to a value in the range -128 to +127. The operation modifies the Carry flag as result of the unsigned addition.

The **ADD** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

In implicit addressing mode, the **ADD** instruction requires no operands:

```
ADD
```

In immediate addressing mode, the **ADD** instruction instruction requires one operand:

```
ADD #number
```

The number sign or hash (**#**) is required to specify the immediate value.

Table 11 describes the attributes of the **ADD** instruction.

Table 11. ADD Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
ADD	50	1	4	Modified
ADD #*number*	58	2	3	Modified

Example:

The following example performs the operation 2+3+4.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
add                ; (5, -, -)
add    #4          ; (9, -, -)
```

The value of TOS after the first **ADD** instruction is 5 because 2+3 = 5. The **ADD #4** instruction adds 4 to TOS (4+5), so that TOS then contains 9. The value of the Carry flag is modified (cleared) after each of the add instructions because neither unsigned addition requires an extra carry bit.

ADD_R (Add and Return)

The **ADD_R** instruction adds two numbers and performs a return-from-call operation. The **ADD_R** instruction uses the implicit addressing mode. The **ADD_R** instruction retrieves both TOS and NEXT from the data stack and adds them together. TOS and NEXT are consumed, and the result is placed in TOS. The operation modifies the Carry flag as result of the unsigned addition. The return-from-call operation loads the instruction pointer (IP) with the return address from the return stack.

The **ADD_R** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **ADD_R** instruction requires no operands:

```
ADD_R
```

Table 12 describes the attributes of the **ADD_R** instruction.

Table 12. ADD_R Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
ADD_R	5C	1	7	Modified

Example:

The following example performs the operation 2+3 and performs the return-from-call operation. The example assumes that the return stack contains the address of the caller to which the **ADD_R** instruction returns.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
add_r              ; (5, -, -)
```

The value of TOS after this code executes is 5 because 2+3 = 5. The value of the Carry flag is modified (cleared) because the unsigned addition did not require an extra carry bit. After completing the addition, the **ADD_R** instruction performs the return-from-call operation.

ALLOC (Allocate)

The **ALLOC** instruction allocates a specified number of bytes on the data stack. The specified number of bytes must be in the range 1 to 8. The **ALLOC** instruction uses the immediate addressing mode. The allocated bytes are not initialized and have non-deterministic values.

Recommendation: Use the **_alloc** system function to allocate more than eight bytes on the data stack, or to allocate any number of bytes that will be initialized to zero. See *_alloc (Allocate Stack Space)* on page 162 for more information.

See also *DEALLOC (Deallocate and Return)* on page 82 to deallocate space on the data stack.

The **ALLOC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **ALLOC** instruction requires a single operand to specify the number of bytes to allocate on the data stack:

```
ALLOC  #number
```

The value of *number* must be in the range 1..8. The number sign or hash (**#**) is required to specify the immediate value.

Table 13 describes the attributes of the **ALLOC** instruction.

Table 13. ALLOC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
ALLOC #1	C0	1	3	None
ALLOC #2	C1	1	3	None
ALLOC #3	C2	1	3	None
ALLOC #4	C3	1	3	None
ALLOC #5	C4	1	3	None
ALLOC #6	C5	1	3	None
ALLOC #7	C6	1	3	None
ALLOC #8	C7	1	3	None

Example:

The following example allocates two bytes on the data stack. The resulting values of TOS and NEXT are not specified.

```
alloc #2            ; (a, b)
```

AND (And)

The **AND** instruction performs a logical AND for two numbers. The **AND** instruction uses one of two addressing modes:

- In implicit addressing mode, the **AND** instruction retrieves both TOS and NEXT from the data stack and performs a bitwise logical AND for them. TOS and NEXT are consumed, and the result is placed in TOS. The operation clears the Carry flag.
- In immediate addressing mode, the **AND** instruction (**AND #*number***) performs a bitwise logical AND for a specified constant *number* with the value of TOS, and the result is placed in TOS. The value of *number* must resolve at link time to a value in the range -128 to +127. The operation clears the Carry flag.

The **AND** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

In implicit addressing mode, the **AND** instruction requires no operands:

```
AND
```

In immediate addressing mode, the **AND** instruction requires one operand:

```
AND #number
```

The number sign or hash (**#**) is required to specify the immediate value.

Table 14 describes the attributes of the **AND** instruction.

Table 14. AND Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
AND	51	1	4	Cleared
AND #*number*	59	2	3	Cleared

Example:

The following example performs the operation 2 AND 3 AND 4.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
and                ; (2, -, -)
and    #4          ; (0, -, -)
```

The value of TOS after the first **AND** instruction is 2 because 2 AND 3 = 2. The **AND #4** instruction performs a logical AND of 4 with TOS (4 AND 2), so that TOS then contains 0.

AND_R (And and Return)

The **AND_R** instruction performs a logical AND for two numbers and performs a return-from-call operation. The **AND_R** instruction uses the implicit addressing mode. The **AND_R** instruction retrieves both TOS and NEXT from the data stack and performs a bitwise logical AND for them. TOS and NEXT are consumed, and the result is placed in TOS. The operation clears the Carry flag. The return-from-call operation loads the instruction pointer (IP) with the return address from the return stack.

The **AND_R** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **AND_R** instruction requires no operands:

```
AND_R
```

Table 15 describes the attributes of the **AND_R** instruction.

Table 15. AND_R Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
AND_R	5D	1	7	Cleared

Example:

The following example performs the operation 2 AND 3 and performs the return-from-call operation. The example assumes that the return stack contains the address of the caller to which the **AND_R** instruction returns.

```
pushs  #2           ; (2, -, -)
pushs  #3           ; (3, -, -)
and_r               ; (2, -, -)
```

The value of TOS after this code executes is 2 because 2 AND 3 = 2. After completing the logical and operation, the **AND_R** instruction performs the return-from-call operation.

BR (Branch)

The **BR** instruction performs an unconditional branch. The **BR** instruction uses the relative addressing mode. The **BR** instruction branches forward or backward with a signed relative displacement of -126 to +129 bytes. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **BR** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *BRF (Branch Far)* on page 72 for a longer unconditional branch and *SBR (Short Branch)* on page 115 for a shorter unconditional forward branch.

The **BR** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BR** instruction requires a single operand to specify the displacement or label:

```
BR displacement
BR label
```

Table 16 describes the attributes of the **BR** instruction.

Table 16. BR Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BR	71	2	2	None

Example:

The following example shows two functions that share a common set of code. The **BR** instruction branches unconditionally from the end of one function to the `Common` label to run the common code.

```
Sub_B    ...                ; subroutine B
         br Common
Sub_A    ...                ; subroutine A
Common   ...                ; common code
         ret                ; return to caller
```

BRC (Branch If Carry)

The **BRC** instruction performs a conditional branch if the Carry flag is set. The **BRC** instruction uses the relative addressing mode. The **BRC** instruction branches forward or backward with a signed relative displacement of -126 to +129 bytes if the Carry flag is set; otherwise, program operation continues with the next instruction. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **BRC** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *BRNC (Branch If Not Carry)* on page 73.

The **BRC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BRC** instruction requires a single operand to specify the displacement or label:

```
BRC displacement
BRC label
```

Table 17 describes the attributes of the **BRC** instruction.

Table 17. BRC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BRC	33	2	2	Used

Example:

The following example assumes that TOS and NEXT have data values to be added. The **ADD** instruction adds them and sets the Carry flag for the unsigned addition. The **BRC** instruction branches to the `Ovflw` label if the Carry flag is set, otherwise the program continues. The **BR** instruction ensures that the code at the `Ovflw` label is only run when needed.

```
          add                ; (result, -, -)
          brc Ovflw
          ...
          br  Common
Ovflw     ...                ; handle Carry set condition
Common    ...
```

BRF (Branch Far)

The **BRF** instruction performs an unconditional branch. The **BRF** instruction uses the relative addressing mode. The **BRF** instruction branches to a specified absolute address. The *address* expression can resolve at link time to any value within the 64 KB address space.

The Neuron Assembler also supports the use of a label for the *address* expression; manual calculation of the displacement value is not required.

See also *BR (Branch)* on page 70 and *SBR (Short Branch)* on page 115 for shorter unconditional branches.

The **BRF** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BRF** instruction requires a single operand to specify the address or label:

```
BRF address
BRF label
```

Table 18 describes the attributes of the **BRF** instruction.

Table 18. BRF Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BRF	75	3	4	None

Example:

The following example uses the **BRF** instruction to branch into a library function, whose address is unknown until the program is compiled and linked.

```
brf LibRoutine  ;
```

BRNC (Branch If Not Carry)

The **BRNC** instruction performs a conditional branch if the Carry flag is not set. The **BRNC** instruction uses the relative addressing mode. The **BRNC** instruction branches forward or backward with a signed relative displacement of -126 to +129 bytes if the Carry flag is not set; otherwise, program operation continues with the next instruction. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **BRNC** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *BRC (Branch If Carry)* on page 71.

The **BRNC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BRNC** instruction requires a single operand to specify the displacement or label:

```
BRNC displacement
BRNC label
```

Table 19 describes the attributes of the **BRNC** instruction.

Table 19. BRNC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BRNC	32	2	2	Used

Example:

The following example assumes that TOS and NEXT have data values to be added. The **ADD** instruction adds them and sets the Carry flag for the unsigned addition. The **BRNC** instruction branches to the `Ok` label if the Carry flag is not set, otherwise the program continues. The **BR** instruction ensures that the code at the `Ok` label is only run when needed.

```
          add                 ; (result, -, -)
          brnc Ok
          ...                 ; handle Carry set condition
          br   Common
Ok        ...
Common    ...
```

BRNEQ (Branch If Not Equal)

The **BRNEQ** instruction compares TOS with a specified number and performs a conditional branch if they are not equal. The **BRNEQ** instruction uses the relative addressing mode for the branch destination and the immediate addressing mode for the test condition. The **BRNEQ** instruction compares TOS to a constant expression. The expression *number* must resolve at link time to a constant in the range -128 to +127.

After the comparison, the **BRNEQ** instruction branches forward or backward with a signed relative displacement of -125 to +130 bytes if TOS is not equal to *number*; otherwise, program operation continues with the next instruction. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (three less than the actual displacement because the displacement calculation includes the size of the **BRNEQ** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

The **BRNEQ** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BRNEQ** instruction requires two operands: one to specify the number for comparison and one to specify the displacement or label:

```
BRNEQ #number, displacement
BRNEQ #number, label
```

The number sign or hash (**#**) is required to specify the immediate value.

Table 20 describes the attributes of the **BRNEQ** instruction.

Table 20. BRNEQ Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BRNEQ	76	3	4 (if branch is taken) 6 (if branch is not taken)	None

Example:

The following example tests the argument in TOS for the ASCII codes 65 (for 'A'), 66 (for 'B'), and 67 (for 'C'), and calls a related function for each. The example shows one possible assembly implementation of a C-language switch/case construct.

```
   Example  APEXP                ; (argument -- )
```

```
testA    brneq #d'65,testB   ; ({argument})
         callr do_a          ; ( )
         br    exit
testB    brneq #d'66,testC   ; ({argument})
         callr do_b          ; ( )
         br    exit
testC    brneq #d'67,fail    ; ({argument})
         callr do_c          ; ( )
         br    exit
fail                         ; (argument)
         callr report_error  ; ( )
exit     ret                 ; return to caller
```

This example does not show the called functions.

BRNZ (Branch If Not Zero)

The **BRNZ** instruction performs a conditional branch if TOS is not zero. The **BRNZ** instruction uses one of two addressing modes: absolute and relative. The **BRNZ** instruction branches forward or backward with a signed relative displacement of -126 to +129 bytes if TOS is not zero; otherwise, program operation continues with the next instruction. TOS is consumed by this instruction. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **BRNZ** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *SBRNZ (Short Branch If Not Zero)* on page 117 for a shorter conditional branch with the same condition, and *BRZ (Branch* If Zero) on page 77 for a conditional branch with the opposite condition.

The **BRNZ** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BRNZ** instruction requires a single operand to specify the displacement or label:

```
BRNZ displacement
BRNZ label
```

Table 21 describes the attributes of the **BRNZ** instruction.

Table 21. BRNZ Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BRNZ	72	2	4	None

Example:

The following example assumes that the stack already has an array of data values from which we want to process non-zero values and skip zero values.

```
search   brnz numFnd
         br   search         ; skip zeros
numFnd   ...
```

BRZ (Branch If Zero)

The **BRZ** instruction performs a conditional branch if TOS is zero. The **BRZ** instruction uses the relative addressing mode. The **BRZ** instruction branches forward or backward with a signed relative displacement of -126 to +129 bytes if TOS is zero; otherwise, program operation continues with the next instruction. TOS is consumed by this instruction. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **BRZ** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *SBRZ (Short Branch If Zero)* on page 119 for a shorter conditional branch with the same condition, and *BRNZ (Branch If Not Zero)* on page 76 for a conditional branch with the opposite condition.

The **BRZ** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **BRZ** instruction requires a single operand to specify the displacement or label:

```
BRZ displacement
BRZ label
```

Table 22 describes the attributes of the **BRZ** instruction.

Table 22. BRZ Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
BRZ	73	2	4	None

Example:

The following example assumes that the stack already has an array of data values from which we want to process zero values and skip non-zero values.

```
search   brz zeroFnd
         br  search          ; skip non-zeros
zeroFnd  ...
```

CALL (Call Near)

The **CALL** instruction calls a function at the specified near address. The **CALL** instruction uses the (short) absolute addressing mode. The *near-address* expression must resolve at link-time to an absolute address in the range h'0000 to h'1FFF. Therefore, this instruction can only be used to call functions that reside in the near area.

See Chapter 8, *System-Provided Functions*, on page 159, for a description of system functions that you can use with this instruction.

The **CALL** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **CALL** instruction requires a single operand to specify the near address:

```
CALL near_address
```

Table 23 describes the attributes of the **CALL** instruction.

Table 23. CALL Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
CALL	00 to 1F	2	6	None

Example:

The following example calls the **_add_8_16f** system function to add an 8-bit value with a 16-bit value.

```
call _add_8_16f  ;
```

CALLF (Call Far)

The **CALLF** instruction calls a function at the specified absolute address. The **CALLF** instruction uses the absolute addressing mode. The *address* expression can resolve at link time to any value within the 64 KB address space.

Recommendation: For calls to addresses in the lower 8 KB of the address space, use the near-address **CALL** instruction. For calls to nearby functions, use the **CALLR** instruction.

The **CALLF** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **CALLF** instruction requires a single operand to specify the far address:

```
CALLF address
```

Table 24 describes the attributes of the **CALLF** instruction.

Table 24. CALLF Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
CALLF	77	3	7	None

Example:

The following example calls a function named *%PTRINC*. This example assumes that the called function resides at a high memory address.

```
            callf %PTRINC       ; calculate next ptr bytes
            ...
  %PTRINC   APEXP               ; function body
            ...
            ret                 ; return to caller
```

CALLR (Call Relative)

The **CALLR** instruction calls a function at the specified forward or backward signed relative displacement of -126 to +129 bytes. The **CALLR** instruction uses the (long) relative addressing mode. The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **CALLR** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

The **CALLR** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **CALLR** instruction requires a single operand to specify the displacement:

```
CALLR displacement
```

Table 25 describes the attributes of the **CALLR** instruction.

Table 25. CALLR Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
CALLR	74	2	5	None

Example:

The following example calls a local function named *myFilter*. This example assumes that the called function is a nearby function.

```
          callr myFilter
          ...
myFilter  ...
          ret             ; return to caller
```

DBRNZ (Decrement and Branch If Not Zero)

The **DBRNZ** instruction performs two functions: it decrements the unsigned 8-bit value on the top of the return stack and performs a conditional branch. The **DBRNZ** instruction uses the indirect addressing mode for the comparison and the relative addressing mode for the branch operation. If the result of the decrement is not zero, the instruction branches to the destination indicated by the *displacement* value, which must be in the range -126 to +129. If the value on the top of the return stack becomes zero, the **DBRNZ** instruction drops the value from the return stack and continues processing with the next instruction. TOS is not affected by this instruction.

The *displacement* expression must resolve at link time to a value in the range -128 to +127 (two less than the actual displacement because the displacement calculation includes the size of the **DBRNZ** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

The **DBRNZ** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **DBRNZ** instruction requires a single operand to specify the displacement:

```
DBRNZ displacement
```

Table 26 describes the attributes of the **DBRNZ** instruction.

Table 26. DBRNZ Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
DBRNZ	70	2	5	None

Example:

The following example calls a function named *do_it* repeatedly, until the value *n* on the return stack is zero. Thus, this example implements a `do until` loop construct. This example assumes that the called function resides at a high memory address or is a system function.

```
Example  APEXP              ; ( n -- )
         poppush            ; ( ) R(n)
loop_body                   ; ( ) R(n)
         callf do_it        ; ( ) R(n)
         dbrnz loop_body    ; ( ) R({n})
         ret                ; ( ) R( )
```

DEALLOC (Deallocate and Return)

The **DEALLOC** instruction deallocates a specified number of bytes from the data stack, and performs a return from call operation. The **DEALLOC** instruction uses the immediate addressing mode. The specified number of bytes must be in the range 1 to 8.

Recommendation: Use the **_dealloc** system function to deallocate more than eight bytes from the data stack. See *_dealloc (Deallocate Stack Space and Return)* on page 163 for more information.

See also *ALLOC (Allocate)* on page 67 to allocate space on the data stack.

The **DEALLOC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **DEALLOC** instruction requires a single operand to specify the number of bytes to allocate on the data stack:

```
DEALLOC   #number
```

The value of *number* must be in the range 1..8 The number sign or hash (**#**) is required to specify the immediate value.

Table 27 describes the attributes of the **DEALLOC** instruction.

Table 27. DEALLOC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
DEALLOC #1	7F	1	6	None
DEALLOC #2	7E	1	6	None
DEALLOC #3	7D	1	6	None
DEALLOC #4	7C	1	6	None
DEALLOC #5	7B	1	6	None
DEALLOC #6	7A	1	6	None
DEALLOC #7	79	1	6	None
DEALLOC #8	78	1	6	None

Example:

The following example allocates two bytes on the data stack as part of a function, then at the end of the function, deallocates the two bytes from the data stack and performs a return from call operation.

```
myFunction
         alloc   #2          ; ((2))
         ...                 ; function body
         dealloc #2          ; ( -- )
```

DEC (Decrement)

The **DEC** instruction decrements the TOS value. The **DEC** instruction uses the implicit addressing mode. If the TOS value is zero prior to the decrement, this instruction sets the Carry flag.

The **DEC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **DEC** instruction requires no operands:

```
DEC
```

Table 28 describes the attributes of the **DEC** instruction.

Table 28. DEC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
DEC	3F	1	2	Modified

Example:

The following example calls assumes that TOS contains a loop-control variable. The **DEC** instruction decrements TOS. When TOS becomes zero, the **BRZ** instruction branches to the `Done` label. As long as TOS is not zero, the **BR** instruction branches to the `Loop` label. Thus, this example implements a `for` loop construct.

```
Loop    ...                 ; loop body
        dec                 ; ( n )
        brz Done
        br  Loop
Done    ...
        ret                 ; return to caller
```

DIV (Divide)

The **DIV** instruction performs integer division. The **DIV** instruction uses the implicit addressing mode. The instruction divides the unsigned integer dividend value in NEXT by the unsigned integer divisor value in TOS. The quotient result is placed in TOS. The remainder, if any, is placed in NEXT. Thus, this instruction implements the following equation:

$$\frac{NEXT}{TOS} \Rightarrow TOS \text{ , with remainder in NEXT}$$

Division by zero has the following effects:

- The quotient in TOS is set to 0xFF.
- The remainder in NEXT is unchanged from the value of the dividend.
- A level 3 exception (subtrap 1), if enabled, is fired. This exception does not cause a device reset.

Recommendation: Use the **_div16** or **_div16s** system functions to divide unsigned or signed 16-bit numbers, and use the **_div8s** system function to divide a signed 8-bit number. See *_div16 (Divide, 16 Bit)* on page 164; *_div16s (Divide Signed, 16 Bit)* on page 164; and *_div8s (Divide Signed, 8 Bit)* on page 165.

The **DIV** instruction applies only to Series 5000 devices.

Syntax:

The **DIV** instruction requires no operands:

```
DIV
```

Table 29 describes the attributes of the **DIV** instruction.

Table 29. DIV Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
DIV	ED	1	14	None

Example:

The following example performs the division 7/3. Before the **DIV** instruction executes, TOS contains 7 and NEXT contains 3. After the **DIV** instruction executes, TOS contains 2 (7/3 = 2, with remainder) and NEXT contains 1 (the remainder).

```
pushs  #3          ; (3, -, -)
pushs  #7          ; (7, 3, -)
div                ; (2, 1, -)
```

DROP (Drop from Stack)

The **DROP** instruction drops a value from one of the stacks:

- The **DROP [RSP]** instruction drops the top of the return stack
- The **DROP TOS** instruction drops TOS
- The **DROP NEXT** instruction drops NEXT

The **DROP** instruction uses the direct addressing mode.

The **DROP** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **DROP** instruction requires a single operand to specify which stack element to drop:

```
DROP [RSP]
DROP TOS
DROP NEXT
```

Table 30 describes the attributes of the **DROP** instruction.

Table 30. DROP Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
DROP [RSP]	F6	1	2	None
DROP TOS	E4	1	3	None
DROP NEXT	E5	1	2	None

Example:

The following example calls a function named *clearFlags*, which requires three stack arguments. The function consumes one of the arguments, but does not consume the others. This example uses the **DROP TOS** instruction twice to clear the remaining arguments from the data stack.

```
          push #MSG_CLR       ; ( flags, *msg_p, len )
          call clearFlags     ; ( *msg_p, len, - )
          drop TOS            ; ( len, -- )
          drop TOS            ; ( -- )
          ...
clearFlags
          ...
          ret                 ; return to caller
```

DROP_R (Drop from Stack and Return)

The **DROP_R** instructions drop a value from the data stack and perform a return from call operation:

- The **DROP_R TOS** instruction drops TOS and performs a return from call
- The **DROP_R NEXT** instruction drops NEXT and performs a return from call

The **DROP_R** instruction uses the direct addressing mode.

Note that the **DROP_R TOS** instruction is equivalent to the **DEALLOC #1** instruction.

The **DROP_R** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **DROP** instruction requires a single operand to specify which stack element to drop:

```
DROP_R TOS
DROP_R NEXT
```

Table 31 describes the attributes of the **DROP_R** instruction.

Table 31. DROP_R Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
DROP_R TOS	7F	1	6	None
DROP_R NEXT	F5	1	5	None

Example:

The following example implements a function named *set_mode*, which uses conditional compilation to perform different tasks for different modes. If the mode `MODE1` is defined, the function consumes the TOS value by using the **POP** instruction. If the mode `MODE1` is not defined, the function uses the **DROP_R TOS** instruction to drop the mode argument from the data stack and return to the function's caller.

```
set_mode                    ; (mode)
IFDEF MODE1
      ; Limit to 0 or 1
      and  #h'1
      pop  _mode            ; ()
      ret                   ; return to caller
ELSE
      drop_r tos            ; ()
ENDIF
```

INC (Increment)

The **INC** instruction can increment the TOS value or the value of a specified pointer register. The **INC** instruction uses the pointer direct addressing mode. When incrementing TOS, if the TOS value is zero after the increment, this instruction sets the Carry flag. When incrementing a pointer register, the *pointer-register* expression must be in the range 0 to 3.

The **INC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **INC** instruction for the TOS requires no operands:

```
INC
```

The **INC** instruction for a pointer register requires a single operand to specify the pointer register:

```
INC [0]
INC [1]
INC [2]
INC [3]
```

Table 32 describes the attributes of the **INC** instruction.

Table 32. INC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
INC	3E	1	2	Modified
INC [0]	34	1	6	None
INC [1]	35	1	6	None
INC [2]	36	1	6	None
INC [3]	37	1	6	None

Example:

The following example calls assumes that TOS contains a loop-control variable with a negative value. The **INC** instruction increments TOS. When TOS becomes zero and Carry flag is set, the **BRC** instruction branches to the `Done` label. As long as the Carry flag is not set, the **BR** instruction branches to the `Loop` label. Thus, this example implements a `for` loop construct.

```
Loop    ...                 ; loop body
        inc                 ; ( n )
        brc Done
        br  Loop
```

```
Done    ...
        ret                 ; return to caller
```

MUL (Multiply)

The **MUL** instruction performs integer multiplication. The **MUL** instruction uses the implicit addressing mode. The instruction multiplies the unsigned integer multiplicand value in NEXT with the unsigned integer multiplier value in TOS. The 16-bit product result is placed in NEXT and TOS. Because the data stack grows towards higher addresses, NEXT holds the most-significant byte (MSB) and TOS holds the least-significant byte (LSB) of the 16-bit value. Thus, this instruction implements the following equation:

$$NEXT * TOS \Rightarrow (NEXT : TOS)$$

where NEXT:TOS represents a 16-bit product.

Recommendation: Use the **_mul16** or **_mul16s** system functions to multiply unsigned or signed 16-bit numbers. See *_mul16 (Multiply, 16 Bit)* on page 183; and *_mul16s (Multiply Signed, 16 Bit)* on page 184.

The **MUL** instruction applies only to Series 5000 devices.

Syntax:

The **MUL** instruction requires no operands:

```
MUL
```

Table 33 describes the attributes of the **MUL** instruction.

Table 33. MUL Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
MUL	EC	1	14	None

Example:

The following example performs the multiplication 18*68 (h'12 * h'44). Before the **MUL** instruction executes, TOS contains 18 (h'12) and NEXT contains 68 (h'44). After the **MUL** instruction executes, TOS contains 200 (h'C8), which are the low 8 bits of the product, and NEXT contains 4 (h'04), which are the high 8 bits of the product. Together, NEXT:TOS contain the 16-bit product, 1224 (h'04C8).

```
pushs  #@LB(68)     ; (h'44, -, -)
pushs  #@LB(18)     ; (h'12, h'44, -)
mul                 ; (h'C8, h'04, -)
```

NOP (No Operation)

The **NOP** instruction performs no operation. The **NOP** instruction uses the implicit addressing mode. This instruction is a one-byte instruction that takes up space and CPU cycles, but does not affect the processor.

Note that this instruction is equivalent to a minimum short branch (forward one byte): **SBR *+1**.

The **NOP** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **NOP** instruction requires no operands:

```
NOP
```

Table 34 describes the attributes of the **NOP** instruction.

Table 34. NOP Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
NOP	20	1	1	None

Example:

The following example uses the **NOP** instruction to adjust the timing cycles required for a specific function. The function causes a delay of 22*count+9 cycles (not including the initial call to the function). The function assumes that the *count* value is already on the data stack.

```
delay_22n_cycles  APEXP
         poppush              ; () [count]
delay_22n_loop
         pushd  [msg_p]
         popd   [msg_p]
         rolc
         rorc
         nop
         dbrnz  delay_22n_loop
         ret                  ; return to caller
```

NOT (Not)

The **NOT** instruction performs a logical NOT operation on the value in TOS. The **NOT** instruction uses the implicit addressing mode. This instruction clears the Carry flag.

The **NOT** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **NOT** instruction requires no operands:

```
NOT
```

Table 35 describes the attributes of the **NOT** instruction.

Table 35. NOT Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
NOT	3D	1	2	Cleared

Example:

The following example defines a *clear_flags* function. This function assumes that flag values to be cleared are in TOS. This function uses the **NOT** instruction to invert the passed flag values, then pushes a saved set of flags onto the stack, performs a logical AND of the inverted flags and the saved flags so that the passed flag values are cleared, then pops the updated flags to memory.

```
%clear_flags  APEXP          ; ( clrMe -- )
         not                 ; ( !clrMe -- )
         push flags          ; ( flags, !clrMe -- )
         and
         pop  flags          ; ( -- )
         ret                 ; return to caller
```

OR (Or)

The **OR** instruction performs a logical OR for two numbers. The **OR** instruction uses one of two addressing modes:

- In implicit addressing mode, the **OR** instruction retrieves both TOS and NEXT from the data stack and performs a bitwise logical OR for them. TOS and NEXT are consumed, and the result is placed in TOS. The operation clears the Carry flag.
- In immediate addressing mode, the **OR** instruction performs a bitwise logical OR for a specified one-byte constant *number* with the value of TOS, and the result is placed in TOS. The value of *number* must resolve at link time to a value in the range -128 to +127. The operation clears the Carry flag.

The **OR** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

In implicit addressing mode, the **OR** instruction requires no operands:

```
OR
```

In immediate addressing mode, the **OR** instruction requires one operand:

```
OR #number
```

The number sign or hash (**#**) is required to specify the immediate value.

Table 36 describes the attributes of the **OR** instruction.

Table 36. OR Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
OR	52	1	4	Cleared
OR #*number*	5A	2	3	Cleared

Example:

The following example performs the operation 2 OR 3 OR 4.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
or                 ; (3, -, -)
or     #4          ; (7, -, -)
```

The value of TOS after the first **OR** instruction is 3 because 2 OR 3 = 3. The **OR #4** instruction performs a logical OR of 4 with TOS (4 OR 3), so that TOS then contains 7.

OR_R (Or and Return)

The **OR_R** instruction performs a logical OR for two numbers and performs a return-from-call operation. The **OR_R** instruction uses the implicit addressing mode. The **OR_R** instruction retrieves both TOS and NEXT from the data stack and performs a bitwise logical OR for them. TOS and NEXT are consumed, and the result is placed in TOS. The operation clears the Carry flag. The return-from-call operation loads the instruction pointer (IP) with the return address from the return stack.

The **OR_R** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **OR_R** instruction requires no operands:

```
OR_R
```

Table 37 describes the attributes of the **OR_R** instruction.

Table 37. OR_R Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
OR_R	5E	1	7	Cleared

Example:

The following example performs the operation 2 OR 3 and performs the return-from-call operation. The example assumes that the return stack contains the address of the caller to which the **OR_R** instruction returns.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
or_r               ; (3, -, -)
```

The value of TOS after this code executes is 3 because 2 OR 3 = 3. After completing the logical OR operation, the **OR_R** instruction performs the return-from-call operation.

POP (Pop from Stack)

The **POP** instruction pops a value from the data stack to a specified location. The **POP** instruction uses one of the following addressing modes:

- In the indirect relative addressing mode, the **POP [*pointer-register*][*offset*]** instruction pops the value from TOS into the location specified by adding (using unsigned arithmetic) the contents of the specified pointer register with the specified offset value. TOS is consumed by this instruction. The value of the *pointer-register* expression must be in the range 0 to 3. The *offset* expression must resolve at link time to a value in the range 0 to 255.
- In the indirect indexed address mode, the **POP [*pointer-register*] [TOS]** instruction pops the value from NEXT into the location specified by adding (using unsigned arithmetic) the contents of the specified pointer register with the value of TOS. This instruction consumes NEXT but preserves TOS. The value of the *pointer-register* expression must be in the range 0 to 3.
- In the direct addressing mode, the **POP *register*** instruction pops the value from TOS into the specified CPU register. The *register* can be DSP, FLAGS, or RSP. Because a **POP** instruction always decrements DSP, the value in DSP after a **POP DSP** instruction is one less than the value popped from TOS. And because the FLAGS register contains the Carry flag, the **POP FLAGS** instruction modifies the Carry flag.
- In the DSP relative addressing mode, the **POP [DSP][*offset*]** instruction pops the value from TOS into the stack location (relative to DSP) specified by *offset*. The value of *offset* must be in the range -1 to -8. Location -1 corresponds to the first byte on the stack below NEXT.
- In the absolute addressing mode, the **POP *address*** instruction pops the value from TOS to the absolute address specified by *address*.
- In the BP relative addressing mode, the **POP !*byte-register*** instruction pops the value from TOS into the specified general-purpose byte register (R0..R15). The *byte-register* expression must resolve at link time to a value in the range 8 to 23 for general-purpose byte registers R0 to R15, respectively.
- In the BP indexed addressing mode, the **POP !TOS** instruction pops the value from NEXT on the data stack into a location specified by adding TOS to the contents of the base page register (BP). This instruction consumes NEXT but preserves TOS.

The **POP** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **POP** instruction requires one or two operands to specify which data stack value to pop and the location into which to store the value:

```
POP [0][offset]
POP [1][offset]
```

```
POP [2][offset]
POP [3][offset]

POP [0][TOS]
POP [1][TOS]
POP [2][TOS]
POP [3][TOS]

POP FLAGS
POP RSP
POP DSP

POP [DSP][-8]
POP [DSP][-7]
POP [DSP][-6]
POP [DSP][-5]
POP [DSP][-4]
POP [DSP][-3]
POP [DSP][-2]
POP [DSP][-1]

POP address

POP !8
POP !9
POP !10
POP !11
POP !12
POP !13
POP !14
POP !15
POP !16
POP !17
POP !18
POP !19
POP !20
POP !21
POP !22
POP !23

POP !TOS
```

The exclamation point (**!**) is required for the **POP !*byte-register*** and **POP !TOS** instructions to specify the displacement relative to the base-page register.

Table 38 describes the attributes of the **POP** instruction.

Table 38. POP Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
POP [0][*offset*]	D8	2	7	None
POP [1][*offset*]	D9	2	7	None
POP [2][*offset*]	DA	2	7	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
POP [3][*offset*]	DB	2	7	None
POP [0][TOS]	DC	1	6	None
POP [1][TOS]	DD	1	6	None
POP [2][TOS]	DE	1	6	None
POP [3][TOS]	DF	1	6	None
POP FLAGS	E0	1	4	Modified
POP RSP	E2	1	4	None
POP DSP	E3	1	4	None
POP [DSP][-8]	F8	1	5	None
POP [DSP][-7]	F9	1	5	None
POP [DSP][-6]	FA	1	5	None
POP [DSP][-5]	FB	1	5	None
POP [DSP][-4]	FC	1	5	None
POP [DSP][-3]	FD	1	5	None
POP [DSP][-2]	FE	1	5	None
POP [DSP][-1]	FF	1	5	None
POP *address*	F7	3	7	None
POP !8	C8	1	4	None
POP !9	C9	1	4	None
POP !10	CA	1	4	None
POP !11	CB	1	4	None
POP !12	CC	1	4	None
POP !13	CD	1	4	None
POP !14	CE	1	4	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
POP !15	CF	1	4	None
POP !16	D0	1	4	None
POP !17	D1	1	4	None
POP !18	D2	1	4	None
POP !19	D3	1	4	None
POP !20	D4	1	4	None
POP !21	D5	1	4	None
POP !22	D6	1	4	None
POP !23	D7	1	4	None
POP !TOS	E6	1	4	None

Examples:

The following example moves the value 23 from TOS into the element following NEXT. The instruction overwrites the destination element and consumes TOS; the remainder of the stack remains unchanged.

```
Example  APEXP                ; ( -- )
         pushs   #1           ; ( 1 )
         pushs   #2           ; ( 2, 1 )
         pushs   #3           ; ( 3, 2, 1 )
         push    #d'23        ; ( d'23, 3, 2, 1 )
         pop     [dsp][-1]    ; ( 3, d'23, 1 )
         dealloc #3           ; ( )
```

The following example clears the general-purpose register R0.

```
R0    EQU   8

Example  APEXP                ; ( -- )
         pushs #0             ; ( 0 )
         pop   !R0            ; ( )
         ret                  ; return to caller
```

The following example function clears the *N*-th byte of the base page. *N* is in the range 0 to 255. This example assumes that *N* is in TOS when calling this function.

```
Example  APEXP                ; ( n -- )
         pushs  #0            ; ( 0, n )
```

```
xch                 ; ( n, 0 )
pop    !tos         ; ( n )
drop_r tos          ; return to caller
```

POPD (Pop Pointer Direct)

The **POPD** instruction pops the two bytes from TOS and NEXT on the data stack into the specified pointer register. The **POPD** instruction uses the pointer direct addressing mode. The *pointer-register* expression must be in the range 0 to 3.

Because the data stack grows towards higher addresses, NEXT holds the most-significant byte (MSB) and TOS holds the least-significant byte (LSB) of the 16-bit value to be popped.

The **POPD** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **POPD** instruction requires one operand to specify the pointer register:

```
POPD [0]
POPD [1]
POPD [2]
POPD [3]
```

Table 39 describes the attributes of the **POPD** instruction.

Table 39. POPD Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
POPD [0]	F0	1	6	None
POPD [1]	F1	1	6	None
POPD [2]	F2	1	6	None
POPD [3]	F3	1	6	None

Example:

The following example preserves the P1 and P2 pointer registers by pushing them onto the stack before performing operations that could affect these registers, then popping them from the stack.

```
P1    EQU  1
P2    EQU  2

; Preserve P1, P2.
      pushd  [P1]
      pushd  [P2]
      ...

      popd   [P1]
      popd   [P2]
      ret                    ; return to caller
```

POPPUSH (Pop from Data Stack and Push onto Return Stack)

The **POPPUSH** instruction pops TOS from the data stack and pushes the value onto the return stack. The **POPPUSH** instruction uses the implicit addressing mode.

See *PUSHPOP (Pop from Return Stack and Push onto Data Stack)* on page 109 for the inverse operation.

The **POPPUSH** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **POPPUSH** instruction requires no operands:

```
POPPUSH
```

Table 40 describes the attributes of the **POPPUSH** instruction.

Table 40. POPPUSH Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
POPPUSH	E7	1	5	None

Example:

The following example calls a function named *do_it* repeatedly, until the value *n* on the return stack is zero. Thus, this example implements a `do until` loop construct. This example assumes that the called function resides at a high memory address or is a system function.

```
Example  APEXP                ; ( n -- )
         poppush              ; ( ) R(n)
loop_body                     ; ( ) R(n)
         callf do_it          ; ( ) R(n)
         dbrnz loop_body      ; ( ) R({n})
         ret                  ; ( ) R( )
```

PUSH (Push onto Stack)

The **PUSH** instructions push a value from a specified location onto the data stack. The **PUSH** instruction uses one of the following addressing modes:

- In the indirect relative addressing mode, the **PUSH [*pointer-register*][*offset*]** instruction pushes the value from the location specified by adding (using unsigned arithmetic) the contents of the specified pointer register with the specified offset value into TOS. The value of the *pointer-register* expression must be in the range 0 to 3. The *offset* expression must resolve at link time to a value in the range 0 to 255.
- In the indirect indexed addressing mode, the **PUSH [*pointer-register*] [TOS]** instruction pushes the value from the location specified by adding (using unsigned arithmetic) the contents of the specified pointer register with the value of TOS into NEXT. This instruction modifies NEXT but preserves TOS. The value of the *pointer-register* expression must be in the range 0 to 3.
- In the direct addressing mode, the **PUSH *register*** instruction pushes the value from the specified CPU register into TOS. The *register* can be DSP, FLAGS, or RSP. Because a **PUSH** instruction always increments DSP, the value in TOS after a **PUSH DSP** instruction is one greater than the original DSP value. And because the FLAGS register contains the Carry flag, the **PUSH FLAGS** instruction allows you to inspect the Carry flag.
- In the DSP relative addressing mode, the **PUSH [DSP][*offset*]** instruction pushes the value from the stack location (relative to DSP) specified by *offset* into TOS. The value of *offset* must be in the range -1 to -8. Location -1 corresponds to the first byte on the stack below NEXT.
- In the absolute addressing mode, the **PUSH *address*** instruction pushes the value from the absolute address specified by *address* into TOS.
- In the direct addressing mode, the **PUSH *register*** instruction pushes the value from the specified register into TOS. The value of *register* must be either TOS or NEXT.
- In the immediate addressing mode, the **PUSH #*number*** instruction pushes the value from the specified number into TOS. The expression *number* must resolve at link time to a value in the range -128 to +127. See also *PUSHS (Push Short)* on page 110.
- In the BP relative addressing mode, the **PUSH !*byte-register*** instruction pushes the value from the specified general-purpose byte register (R0..R15) into TOS. The *byte-register* expression must resolve at link time to a value in the range 8 to 23 for general-purpose byte registers R0 to R15, respectively.
- In the indirect addressing mode, the **PUSH [RSP]** instruction pushes the byte on the top of the return stack into TOS. The byte on top of the return stack is not consumed by this instruction and the return stack pointer (RSP) remains unchanged.
- In the BP indexed addressing mode, the **PUSH !TOS** instruction pushes the value from a location specified by adding TOS to the contents of the

base page register (BP) into NEXT on the data stack. This instruction modifies NEXT but preserves TOS.

The **PUSH** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **PUSH** instruction requires one or two operands to specify which data stack value to pop and the location into which to store the value:

```
PUSH [0][offset]
PUSH [1][offset]
PUSH [2][offset]
PUSH [3][offset]

PUSH [0][TOS]
PUSH [1][TOS]
PUSH [2][TOS]
PUSH [3][TOS]

PUSH FLAGS
PUSH RSP
PUSH DSP

PUSH [DSP][-8]
PUSH [DSP][-7]
PUSH [DSP][-6]
PUSH [DSP][-5]
PUSH [DSP][-4]
PUSH [DSP][-3]
PUSH [DSP][-2]
PUSH [DSP][-1]

PUSH address

PUSH TOS
PUSH NEXT

PUSH #number

PUSH !8
PUSH !9
PUSH !10
PUSH !11
PUSH !12
PUSH !13
PUSH !14
PUSH !15
PUSH !16
PUSH !17
PUSH !18
PUSH !19
PUSH !20
PUSH !21
PUSH !22
PUSH !23
```

```
PUSH [RSP]

PUSH !TOS
```

The number sign or hash (**#**) is required to specify the immediate value for the **PUSH #*number*** instruction. The exclamation point (**!**) is required for the **PUSH !*byte-register*** and **PUSH !TOS** instructions to specify the displacement relative to the base-page register.

Table 41 describes the attributes of the **PUSH** instruction.

Table 41. PUSH Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
PUSH [0][*offset*]	98	2	7	None
PUSH [1][*offset*]	99	2	7	None
PUSH [2][*offset*]	9A	2	7	None
PUSH [3][*offset*]	9B	2	7	None
PUSH [0][TOS]	9C	1	6	None
PUSH [1][TOS]	9D	1	6	None
PUSH [2][TOS]	9E	1	6	None
PUSH [3][TOS]	9F	1	6	None
PUSH FLAGS	A0	1	4	None
PUSH RSP	A2	1	4	None
PUSH DSP	A3	1	4	None
PUSH [DSP][-8]	B8	1	5	None
PUSH [DSP][-7]	B9	1	5	None
PUSH [DSP][-6]	BA	1	5	None
PUSH [DSP][-5]	BB	1	5	None
PUSH [DSP][-4]	BC	1	5	None
PUSH [DSP][-3]	BD	1	5	None
PUSH [DSP][-2]	BE	1	5	None
PUSH [DSP][-1]	BF	1	5	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
PUSH *address*	B7	3	7	None
PUSH TOS	A4	1	3	None
PUSH NEXT	A5	1	4	None
PUSH #*number*	B4	2	4	None
PUSH !8	88	1	4	None
PUSH !9	89	1	4	None
PUSH !10	8A	1	4	None
PUSH !11	8B	1	4	None
PUSH !12	8C	1	4	None
PUSH !13	8D	1	4	None
PUSH !14	8E	1	4	None
PUSH !15	8F	1	4	None
PUSH !16	90	1	4	None
PUSH !17	91	1	4	None
PUSH !18	92	1	4	None
PUSH !19	93	1	4	None
PUSH !20	94	1	4	None
PUSH !21	95	1	4	None
PUSH !22	96	1	4	None
PUSH !23	97	1	4	None
PUSH [RSP]	A1	1	4	None
PUSH !TOS	A6	1	4	None

Examples:

The following example assumes that the stack contains two 16-bit elements, A and B, with A nearest to the top of the stack. The example places a copy of B in TOS and NEXT.

```
Example  APEXP          ; ( a(2), b(2) -- b(2), a(2), b(2) )
         push [dsp][-2]      ; ( msb(b), a(2), b(2) )
         push [dsp][-2]      ; ( lsb(b), a(2), b(2) )
         ret                 ; ( b(2), a(2), b(2) )
```

The following example loads TOS with the contents of the general-purpose register R0.

```
R0       EQU  8

Example  APEXP               ; ( -- r0)
         push  !R0           ; ( r0 )
         ret                 ; return to caller
```

The following example calls the *do_it_n* function *N* times, with *N* >= 0 in TOS. The number of remaining invocations is passed to the *do_it_n* function with each call, which consumes its argument.

```
Example  APEXP               ; ( n -- )
         poppush             ; ( ) R(n)
loop_body                    ; ( ) R(n)
         push [rsp]          ; ( n ) R(n)
         dec                 ; ( n-- ) R(n)
         callf do_it_n       ; ( ) R(n)
         dbrnz loop_body     ; ( ) R({n})
         ret                 ; ( ) R( )
```

The following example loads TOS with the content of *N*-th byte in the base page. *N* is passed into this example function in TOS.

```
Example  APEXP               ; ( n -- v )
         push   !tos         ; ( n, v --  )
         drop_r tos          ; ( v )
```

PUSHD (Push Pointer Direct)

The **PUSHD** instruction pushes two bytes from the specified location into TOS and NEXT on the data stack. The **PUSHD** instruction uses one of the following addressing modes:

- In the pointer direct addressing mode, the **PUSHD [*pointer-register*]** instruction pushes the two bytes from the specified pointer-register into TOS and NEXT. The *pointer-register* expression must be in the range 0 to 3.
- In the immediate addressing mode, the **PUSHD #*expression*** instruction pushes the 16-bit value of *expression* into TOS and NEXT.
- In the immediate addressing mode, the **PUSHD #*msb*,#*lsb*** instruction pushes an 8-bit most-significant byte (*msb*) value into NEXT and an 8-bit least-significant byte (*lsb*) value into TOS. *msb* and *lsb* must each resolve at link time to a value in the range 0 to 255. This instruction is equivalent to the **PUSHD #*expression*** instruction.

Because the data stack grows towards higher addresses, NEXT holds the most-significant byte (MSB) and TOS holds the least-significant byte (LSB) of the 16-bit value that is pushed onto the stack.

The **PUSHD** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **PUSHD** instruction requires one or two operands to specify the location from which to push the data:

```
PUSHD [0]
PUSHD [1]
PUSHD [2]
PUSHD [3]

PUSHD #expression

PUSHD #msb,#lsb
```

The number sign or hash (**#**) is required to specify the immediate value for the **PUSHD #*expression*** and **PUSHD #*msb*,#*lsb*** instructions.

Table 42 describes the attributes of the **PUSHD** instruction.

Table 42. PUSHD Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
PUSHD [0]	B0	1	6	None
PUSHD [1]	B1	1	6	None
PUSHD [2]	B2	1	6	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
PUSHD [3]	B3	1	6	None
PUSHD *#expression*	B5	3	6	None
PUSHD *#msb,#lsb*	B5	3	6	None

Example:

The following example preserves the P1 and P2 pointer registers by pushing them onto the stack before performing operations that could affect these registers, then popping them from the stack.

```
P1    EQU  1
P2    EQU  2

; Preserve P1, P2.
      pushd  [P1]
      pushd  [P2]
      ...

      popd   [P1]
      popd   [P2]
      ret                    ; return to caller
```

PUSHPOP (Pop from Return Stack and Push onto Data Stack)

The **PUSHPOP** instruction pops the byte from the top of the return stack and pushes it into TOS on the data stack. The **PUSHPOP** instruction uses the implicit addressing mode.

You can also use the **PUSH [RSP]** instruction to save the value at the top of the return stack without removing it.

See *POPPUSH (Pop from Data Stack and Push onto Return Stack)* on page 101 for the inverse operation.

The **PUSHPOP** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **PUSHPOP** instruction requires no operands:

```
PUSHPOP
```

Table 43 describes the attributes of the **PUSHPOP** instruction.

Table 43. PUSHPOP Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
PUSHPOP	A7	1	5	None

Example:

The following example uses the **POPPUSH** instruction to save the contents of the top of the stack so that it can store other values onto the stack below the top values. It then uses the **PUSHPOP** instruction to restore the values to the top of the stack. The comments show the contents of the data stack and the return stack for each operation.

```
poppush                 ; (A--)  R(B--)
poppush                 ; (--)  R(A, B--)
pushd   [C]             ; (A --)  R(A, B --)
pushd   [D]             ; (D, C --)  R(A, B --)
pushd   [E]             ; (E, D, C --)  R(A, B --)
pushpop                 ; (A, E, D, C -- )  R( B -- )
pushpop                 ; (B, A, E, D, C -- )  R( -- )
```

PUSHS (Push Short)

The **PUSHS** instruction pushes the value of the specified operand *number* into TOS. The **PUSHS** instruction uses the immediate addressing mode. The expression *number* must be in the range 0 to 7.

See also *PUSH (Push onto Stack)* on page 102 for information about the **PUSH #*number*** instruction.

The **PUSHS** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **PUSHS** instruction requires a single operand to specify the number:

```
PUSHS #number
```

The value of *number* must be in the range 0..7. The number sign or hash (**#**) is required to specify the immediate value.

Table 44 describes the attributes of the **PUSHS** instruction.

Table 44. PUSHS Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
PUSHS #0	80	1	4	None
PUSHS #1	81	1	4	None
PUSHS #2	82	1	4	None
PUSHS #3	83	1	4	None
PUSHS #4	84	1	4	None
PUSHS #5	85	1	4	None
PUSHS #6	86	1	4	None
PUSHS #7	87	1	4	None

Example:

The following example performs the operation 2 AND 3 by pushing the immediate values 2 and 3 onto the stack.

```
pushs  #2          ; (2, -, -)
pushs  #3          ; (3, 2, -)
and                ; (2, -, -)
```

RET (Return from Call)

The **RET** instruction performs a return-from-call operation. The **RET** instruction uses the implicit addressing mode.

The **RET** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **RET** instruction requires no operands:

```
RET
```

Table 45 describes the attributes of the **RET** instruction.

Table 45. RET Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
RET	31	1	4	None

Example:

The following example calls a function named *clearFlags*. After the function completes, the **RET** instruction returns from the function, and the function continues operation with the instruction that follows **CALL** instruction.

```
            call clearFlags     ; ( )
            ...
            br  somewhere
   clearFlags
            ...
            ret                 ; return to caller
```

ROLC (Rotate Left through Carry)

The **ROLC** instruction rotates the byte in TOS one bit to the left, using the Carry flag as an input bit. The **ROLC** instruction uses the implicit addressing mode. Bit 0 is loaded from the Carry flag. After the operation completes, the Carry flag contains the value from bit 7.

Figure 9 shows the operation of the **ROLC** instruction.

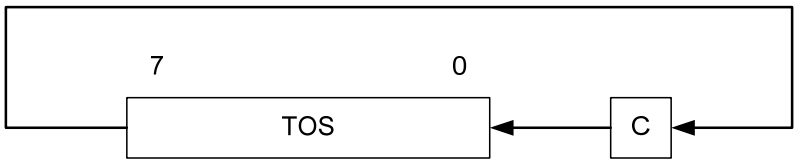

Figure 9. The ROLC Instruction

The **ROLC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **ROLC** instruction requires no operands:

```
ROLC
```

Table 46 describes the attributes of the **ROLC** instruction.

Table 46. ROLC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
ROLC	39	1	2	Used, modified

Example:

The following example pushes the number 150 (h'96) onto the data stack, and then uses the **ROLC** instruction to rotate the TOS value left through the Carry flag. This example assumes that the Carry flag is not set (has value 0). After the **ROLC** instruction completes, TOS has the value 44 (h'2C) and the Carry flag is set (has value 1 from bit 7 of the original h'96 in TOS).

```
push    #d'150      ; ( d'150, -- )
rolc                ; ( d'44, -- )
```

RORC (Rotate Right through Carry)

The **RORC** instruction rotates the byte in TOS one bit to the right, using the Carry flag as an input bit. The **RORC** instruction uses the implicit addressing mode. Bit 7 is loaded from the Carry flag. After the operation completes, the Carry flag contains the value from bit 0.

Figure 10 shows the operation of the **RORC** instruction.

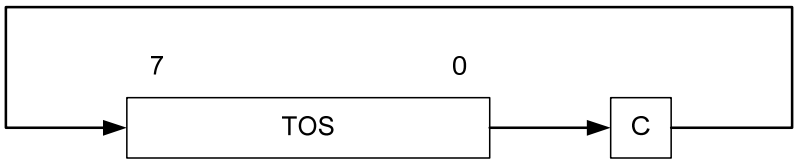

Figure 10. The RORC Instruction

The **RORC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **RORC** instruction requires no operands:

```
RORC
```

Table 47 describes the attributes of the **RORC** instruction.

Table 47. RORC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
RORC	38	1	2	Used, modified

Example:

The following example pushes the number 150 (h'96) onto the data stack, and then uses the **RORC** instruction to rotate the TOS value right through the Carry flag. This example assumes that the Carry flag is not set (has value 0). After the **RORC** instruction completes, TOS has the value 75 (h'4B) and the Carry flag is not set (has value 0 from bit 0 of the original h'96 in TOS).

```
push    #d'150      ; ( d'150, -- )
rorc                ; ( d'75, -- )
```

SBC (Subtract with Carry)

The **SBC** instruction subtracts two numbers, and uses the value of the Carry flag as input to the subtrahend. The **SBC** instruction uses the direct addressing mode. The **SBC** instruction retrieves both TOS and NEXT from the data stack, adds the value of the Carry flag to TOS, and then subtracts TOS (the subtrahend) from NEXT (the minuend). TOS and NEXT are consumed, and the result (the difference) is placed in TOS. The operation modifies the Carry flag as result of the unsigned subtraction: if the value of TOS plus the value of the Carry flag exceeds the value of NEXT (indicating that the result would be negative), the **SBC** instruction sets the Carry flag; otherwise the instruction clears the Carry flag.

The **SBC** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SBC** instruction requires two operands:

```
SBC NEXT,TOS
```

Table 48 describes the attributes of the **SBC** instruction.

Table 48. SBC Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBC NEXT,TOS	56	1	4	Used, modified

Example:

The following example performs the operation 3-2. The example assumes that the Carry flag is not set.

```
pushs  #3           ; (3, -, -)
pushs  #2           ; (2, 3, -)
sbc    next,tos     ; (1, -, -)
```

The value of TOS after this code executes is 1 because (3+Carry)-2 = 1. In this case, the value of the Carry flag is used for the subtraction, and then modified (cleared) because the unsigned subtraction did not require an extra carry bit.

SBR (Short Branch)

The **SBR** instruction performs an unconditional branch. The **SBR** instruction uses the (short) relative addressing mode. The **SBR** instruction branches forward with an unsigned relative displacement of 1 to 16 bytes. The *displacement* expression must resolve at link time to a value in the range 0 to 15 (one less than the actual displacement because the displacement calculation includes the size of the **SBR** operation itself).

Note that the minimum short branch (forward one byte), **SBR *+1**, is equivalent to a no operation (**NOP**) instruction.

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *BR (Branch)* on page 70 and *BRF (Branch Far)* on page 72 for longer unconditional branches.

The **SBR** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SBR** instruction requires a single operand to specify the displacement or label:

```
SBR displacement
SBR label
```

Table 49 describes the attributes of the **SBR** instruction. In the table, the asterisk (*****) specifies the current value of the instruction pointer (IP).

Table 49. SBR Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBR *+1	20	1	1	None
SBR *+2	21	1	1	None
SBR *+3	22	1	1	None
SBR *+4	23	1	1	None
SBR *+5	24	1	1	None
SBR *+6	25	1	1	None
SBR *+7	26	1	1	None
SBR *+8	27	1	1	None
SBR *+9	28	1	1	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBR *+10	29	1	1	None
SBR *+11	2A	1	1	None
SBR *+12	2B	1	1	None
SBR *+13	2C	1	1	None
SBR *+14	2D	1	1	None
SBR *+15	2E	1	1	None
SBR *+16	2F	1	1	None

Example:

The following example shows two functions that share a common set of code. The **SBR** instruction branches unconditionally from the end of one function to the `Common` label to run the common code. This example assumes that the code at label `Sub_A` requires fewer than 16 bytes.

```
Sub_B    ...                ; subroutine B
         sbr Common
Sub_A    ...                ; subroutine A
Common   ...                ; common code
         ret                ; return to caller
```

SBRNZ (Short Branch If Not Zero)

The **SBRNZ** instruction performs a conditional branch if TOS is not zero. The **SBRNZ** instruction uses the (short) relative addressing mode. The **SBRNZ** instruction branches forward with an unsigned relative displacement of 1 to 16 bytes if TOS is not zero; otherwise, program operation continues with the next instruction. TOS is consumed by this instruction. The *displacement* expression must resolve at link time to a value in the range 0 to 15 (one less than the actual displacement because the displacement calculation includes the size of the **SBRNZ** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *BRNZ (Branch If Not Zero)* on page 76 for a longer conditional branch with the same condition, and *SBRZ (Short Branch If Zero)* on page 119 for a short conditional branch with the opposite condition.

The **SBRNZ** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SBRNZ** instruction requires a single operand to specify the displacement or label:

```
SBRNZ displacement
SBRNZ label
```

Table 50 describes the attributes of the **SBRNZ** instruction. In the table, the asterisk (*****) specifies the current value of the instruction pointer (IP).

Table 50. SBRNZ Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBRNZ *+1	60	1	3	None
SBRNZ *+2	61	1	3	None
SBRNZ *+3	62	1	3	None
SBRNZ *+4	63	1	3	None
SBRNZ *+5	64	1	3	None
SBRNZ *+6	65	1	3	None
SBRNZ *+7	66	1	3	None
SBRNZ *+8	67	1	3	None
SBRNZ *+9	68	1	3	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBRNZ *+10	69	1	3	None
SBRNZ *+11	6A	1	3	None
SBRNZ *+12	6B	1	3	None
SBRNZ *+13	6C	1	3	None
SBRNZ *+14	6D	1	3	None
SBRNZ *+15	6E	1	3	None
SBRNZ *+16	6F	1	3	None

Example:

The following example assumes that the stack already has an array of data values from which we want to process non-zero values and skip zero values.

```
search   sbrnz numFnd
         sbr    search       ; skip zeros
numFnd   ...
```

SBRZ (Short Branch If Zero)

The **SBRZ** instruction performs a conditional branch if TOS is zero. The **SBRZ** instruction uses the (short) relative addressing mode. The **SBRZ** instruction branches forward with an unsigned relative displacement of 1 to 16 bytes if TOS is zero; otherwise, program operation continues with the next instruction. TOS is consumed by this instruction. The *displacement* expression must resolve at link time to a value in the range 0 to 15 (one less than the actual displacement because the displacement calculation includes the size of the **SBRZ** operation itself).

The Neuron Assembler also supports the use of a label for the *displacement* expression; manual calculation of the displacement value is not required.

See also *BRZ (Branch If Zero)* on page 77 for a longer conditional branch with the same condition, and *SBRNZ (Short Branch If Not Zero)* on page 117 for a short conditional branch with the opposite condition.

The **SBRZ** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SBRZ** instruction requires a single operand to specify the displacement or label:

```
SBRZ displacement
SBRZ label
```

Table 51 describes the attributes of the **SBRZ** instruction. In the table, the asterisk (*****) specifies the current value of the instruction pointer (IP).

Table 51. SBRZ Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBRZ *+1	40	1	3	None
SBRZ *+2	41	1	3	None
SBRZ *+3	42	1	3	None
SBRZ *+4	43	1	3	None
SBRZ *+5	44	1	3	None
SBRZ *+6	45	1	3	None
SBRZ *+7	46	1	3	None
SBRZ *+8	47	1	3	None
SBRZ *+9	48	1	3	None

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SBRZ *+10	49	1	3	None
SBRZ *+11	4A	1	3	None
SBRZ *+12	4B	1	3	None
SBRZ *+13	4C	1	3	None
SBRZ *+14	4D	1	3	None
SBRZ *+15	4E	1	3	None
SBRZ *+16	4F	1	3	None

Example:

The following example assumes that the stack already has an array of data values from which we want to process zero values and skip non-zero values.

```
search   sbrz zeroFnd
         sbr   search        ; skip non-zeros
zeroFnd  ...
```

SHL (Shift Left)

The **SHL** instruction shifts the byte in TOS one bit to the left. The **SHL** instruction uses the implicit addressing mode. This instruction moves a zero into bit 0 of TOS, and discards bit 7. This instruction also clears the Carry flag.

Figure 11 shows the operation of the **SHL** instruction.

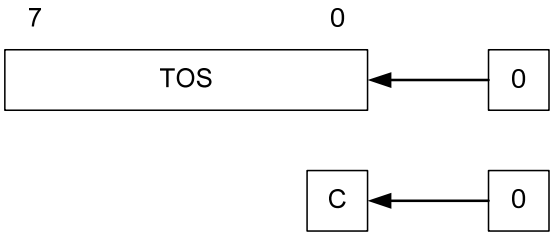

Figure 11. The SHL Instruction

The **SHL** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SHL** instruction requires no operands:

```
SHL
```

Table 52 describes the attributes of the **SHL** instruction.

Table 52. SHL Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SHL	3B	1	2	Cleared

Example:

The following example pushes the number 150 (h'96) onto the data stack, and then uses the **SHL** instruction to shift the TOS value left. After the **SHL** instruction completes, TOS has the value 44 (h'2C) and the Carry flag is cleared.

```
push    #d'150      ; ( d'150, -- )
shl                 ; ( d'44, -- )
```

SHLA (Shift Left Arithmetically)

The **SHLA** instruction shifts the byte in TOS one bit to the left arithmetically. The **SHLA** instruction uses the implicit addressing mode. This instruction moves a zero into bit 0 of TOS, and stores bit 7 in the Carry flag.

Figure 12 shows the operation of the **SHLA** instruction.

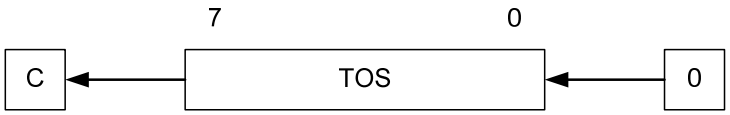

Figure 12. The SHLA Instruction

The **SHLA** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SHLA** instruction requires no operands:

```
SHLA
```

Table 53 describes the attributes of the **SHLA** instruction.

Table 53. SHLA Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SHLA	30	1	2	Modified

Example:

The following example pushes the number 150 (h'96) onto the data stack, and then uses the **SHLA** instruction to shift the TOS value left arithmetically. After the **SHLA** instruction completes, TOS has the value 44 (h'2C) and the Carry flag is set to 1 (the previous value of bit 7 in TOS).

```
push    #d'150      ; ( d'150, -- )
shla                ; ( d'44, -- )
```

SHR (Shift Right)

The **SHR** instruction shifts the byte in TOS one bit to the right. The **SHR** instruction uses the implicit addressing mode. This instruction moves a zero into bit 7 of TOS, and discards bit 0. This instruction also clears the Carry flag.

Figure 13 shows the operation of the **SHR** instruction.

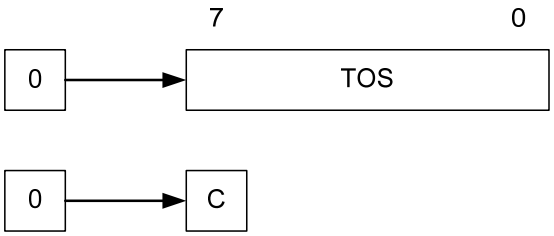

Figure 13. The SHR Instruction

The **SHR** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SHR** instruction requires no operands:

```
SHR
```

Table 54 describes the attributes of the **SHR** instruction.

Table 54. SHR Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SHR	3A	1	2	Cleared

Example:

The following example pushes the number 150 (h'96) onto the data stack, and then uses the **SHR** instruction to shift the TOS value right. After the **SHR** instruction completes, TOS has the value 75 (h'4B) and the Carry flag is cleared.

```
push    #d'150      ; ( d'150, -- )
shr                 ; ( d'75, -- )
```

SHRA (Shift Right Arithmetically)

The **SHRA** instruction shifts the byte in TOS one bit to the right arithmetically. The **SHRA** instruction uses the implicit addressing mode. This instruction moves bit 0 of TOS into the Carry flag, and leaves bit 7 of TOS unchanged.

Figure 14 shows the operation of the **SHRA** instruction.

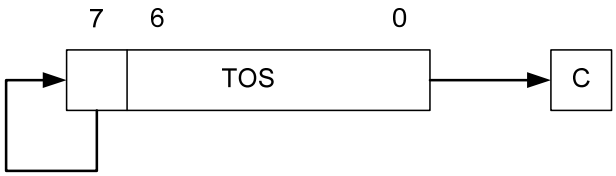

Figure 14. The SHRA Instruction

The **SHRA** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SHRA** instruction requires no operands:

```
SHRA
```

Table 55 describes the attributes of the **SHRA** instruction.

Table 55. SHRA Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SHRA	3C	1	2	Modified

Example:

The following example pushes the number 150 (h'96) onto the data stack, and then uses the **SHRA** instruction to shift the TOS value right arithmetically. After the **SHRA** instruction completes, TOS has the value 203 (h'CB) and the Carry flag is set to 0 (the previous value of bit 0 in TOS).

```
push    #d'150      ; ( d'150, -- )
shra                ; ( d'203, -- )
```

SUB (Subtract)

The **SUB** instruction subtracts two numbers. The **SUB** instruction uses the direct addressing mode, in two forms, subtract NEXT from TOS, and subtract TOS from NEXT:

- The **SUB NEXT,TOS** instruction retrieves both TOS and NEXT from the data stack and subtracts NEXT from TOS. TOS and NEXT are consumed, and the result is placed in TOS. The operation modifies the Carry flag as result of the unsigned subtraction: if the value of TOS exceeds the value of NEXT (indicating that the result would be negative), the **SUB** instruction sets the Carry flag; otherwise the instruction clears the Carry flag.
- The **SUB TOS,NEXT** instruction retrieves both TOS and NEXT from the data stack and subtracts TOS from NEXT. TOS and NEXT are consumed, and the result is placed in TOS. The operation modifies the Carry flag as result of the unsigned subtraction: if the value of NEXT exceeds the value of TOS (indicating that the result would be negative), the **SUB** instruction sets the Carry flag; otherwise the instruction clears the Carry flag.

The **SUB** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **SUB** instruction requires two operands:

```
SUB NEXT,TOS
SUB TOS,NEXT
```

Table 56 describes the attributes of the **SUB** instruction.

Table 56. SUB Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
SUB NEXT,TOS	57	1	4	Modified
SUB TOS,NEXT	55	1	4	Modified

Example:

The following example performs the operation 3-2 and the operation 2-3.

```
pushs  #3          ; (3, -, -)
pushs  #2          ; (2, 3, -)
sub    next,tos    ; (1, -, -)
pushs  #3          ; (3, 1, -)
pushs  #2          ; (2, 3, 1)
sub    tos,next    ; (255, 1, -)
```

The value of TOS after the first subtraction is 1 because 3-2 = 1. In this case, the value of the Carry flag is cleared because the unsigned subtraction did not require an extra carry bit.

The value of TOS after the second subtraction is 255 (h'FF) because 2-3 = -1. In this case, the value of the Carry flag is set because NEXT was greater than TOS and the unsigned subtraction required an extra carry bit.

XCH (Exchange)

The **XCH** instruction exchanges the contents of TOS and NEXT on the data stack. The **XCH** instruction uses the implicit addressing mode.

The **XCH** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **XCH** instruction requires no operands:

```
XCH
```

Table 57 describes the attributes of the **XCH** instruction.

Table 57. XCH Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
XCH	B6	1	4	None

Example:

The following example pushes two numbers onto the data stack, then exchanges their positions on the stack.

```
pushs  #3          ; (3, -, -)
pushs  #2          ; (2, 3, -)
xch                ; (3, 2, -)
```

XOR (Exclusive Or)

The **XOR** instruction performs a logical exclusive OR (XOR) for two numbers. The **OR** instruction uses one of two addressing modes:

- In implicit addressing mode, the **XOR** instruction retrieves both TOS and NEXT from the data stack and performs a bitwise logical XOR for them. TOS and NEXT are consumed, and the result is placed in TOS. The operation clears the Carry flag.
- In immediate addressing mode, the **XOR** instruction performs a bitwise logical XOR for a specified one-byte constant *number* with the value of TOS, and the result is placed in TOS. The value of *number* must resolve at link time to a value in the range -128 to +127. The operation clears the Carry flag.

The **XOR** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

In implicit addressing mode, the **XOR** instruction requires no operands:

```
XOR
```

In immediate addressing mode, the **XOR** instruction requires one operand:

```
XOR #number
```

The number sign or hash (**#**) is required to specify the immediate value.

Table 58 describes the attributes of the **XOR** instruction.

Table 58. XOR Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
XOR	53	1	4	Cleared
XOR #*number*	5B	2	3	Cleared

Example:

The following example performs the operation (2 XOR 3) XOR 4.

```
pushs  #2           ; (2, -, -)
pushs  #3           ; (3, 2, -)
xor                 ; (1, -, -)
xor     #4          ; (5, -, -)
```

The value of TOS after the first **XOR** instruction is 1 because 2 XOR 3 = 1. The **XOR #4** instruction performs a logical XOR of 4 with TOS (4 XOR 1), so that TOS then contains 5.

XOR_R (Exclusive Or and Return)

The **XOR_R** instruction performs a logical exclusive OR (XOR) for two numbers and performs a return-from-call operation. The **XOR_R** instruction uses the implicit addressing mode. The **XOR_R** instruction retrieves both TOS and NEXT from the data stack and performs a bitwise logical XOR for them. TOS and NEXT are consumed, and the result is placed in TOS. The operation clears the Carry flag. The return-from-call operation loads the instruction pointer (IP) with the return address from the return stack.

The **XOR_R** instruction applies to both Series 3100 and Series 5000 devices.

Syntax:

The **XOR_R** instruction requires no operands:

```
XOR_R
```

Table 59 describes the attributes of the **XOR_R** instruction.

Table 59. XOR_R Instruction

Instruction	Hexadecimal Opcode	Instruction Size (Bytes)	CPU Cycles Required	Affect on Carry Flag
XOR_R	5F	1	7	Cleared

Example:

The following example performs the operation 2 XOR 3 and performs the return-from-call operation. The example assumes that the return stack contains the address of the caller to which the **XOR_R** instruction returns.

```
pushs  #2           ; (2, -, -)
pushs  #3           ; (3, 2, -)
xor_r               ; (1, -, -)
```

The value of TOS after this code executes is 3 because 2 XOR 3 = 1. After completing the logical XOR operation, the **XOR_R** instruction performs the return-from-call operation.

7

Neuron Assembler Directives

The Neuron Assembler provides various directives to control the inclusion of additional source files, to control conditional compilation, to control code segments, to manage symbols, to change the default radix, and to control the assembly listing output.

This chapter describes the directives that the Neuron Assembler supports.

Overview of the Assembler Directives

An assembler directive provides information to the Neuron Assembler to control or affect the processing of the remainder of the assembly file. All directives are optional. Many assembly files do not require any directives.

Table 60 lists the directives for the Neuron Assembler, grouped by general function: symbol control and scope control, data allocation, segment control, conditional assembly, and other directives.

Table 60. Neuron Assembler Directives

Symbols and Scope Control	
APEXP, EXPORT	Export symbol
EQU	Define mnemonic for constant
IMPORT	Import external symbol for linkage
INCLUDE	Include assembly source file
LIBRARY	Import function library
Data Allocation	
DATA.B	Define initialized data
RES	Reserve space for uninitialized data
Segment Control	
ORG	Specify location in segment
SEG	Select segment
Conditional Assembly	
ELSE, ELSEIF	Control conditional assembly
ENDIF, IF	
ERROR	
IFDEF, IFNDEF	
Other Directives	
END	End assembly
LIST, NOLIST, PAGE, SUBHEAD	Control listing

RADIX	Specify default numeric base
RESOURCE	Control resource (compiler generated)

The directives have syntax analogous to the assembly instructions described in Chapter 6, *Neuron Assembly Language Instruction Statements*, on page 59. Most directives require arguments, which are similar to the operands of the assembly instructions.

The following sections describe each of the Neuron Assembler directives.

APEXP (Application Symbol Export)

The **APEXP** directive is used to export symbols that interface with the Neuron C application.

An exported symbol must be defined as a label within the assembly file from which it is exported. Exporting the symbol makes it available to the linker. The linker uses exported symbols to resolve references from other object modules. It is possible, and sometimes required, to export and import a symbol within the same assembly file to permit the linker to resolve expressions between separate segments.

See also *EXPORT (Export Symbol)* on page 142.

Syntax:

The **APEXP** directive requires either a label or an argument, or both.

```
label APEXP
      APEXP symbol
      APEXP symbol, symbol, ...
label APEXP symbol
label APEXP symbol, symbol, ...
```

An argument can be a symbol, or it can be a list of symbols separated by commas. The argument specifies the symbols to be exported. If there is no argument for the directive, it must have a label, and the label becomes the exported symbol. If there are both an argument and a label, the argument specifies the symbols to be exported and the label is a standard label.

Examples:

The following example exports the label *%IOToggle* as an external symbol to the Neuron C application:

```
%IOToggle    APEXP
```

The following example exports the symbols *Flag1* and *Flag2* to the Neuron C application:

```
InFlags      APEXP  Flag1, Flag2
```

DATA.B (Reserve Initialized Memory)

The **DATA.B** directive is used to reserve memory and initialize the reserved memory to specific values.

See also *RES (Reserve Uninitialized Memory)* on page 155.

Syntax:

The **DATA.B** directive requires an expression, or a list of expressions separated by commas. A label is optional.

```
        DATA.B expr
        DATA.B expr, expr, ...
label   DATA.B expr
label   DATA.B expr, expr, ...
```

The assembler reserves a block of data with a size, in bytes, that matches the number of expressions in the argument, and initializes each byte to the corresponding expression's value.

The *expr* expression can have one of the following forms:

- A numeric expression with a value in the range 0..255. Each number, which can be a constant of a symbolic expression resolved by the linker, represents one byte of initialized data. For example:

  ```
  DATA.B  h'12, h'34, h'56
  ```

- A double-quoted string argument that represents a number of bytes equal to the number of characters in the string (there is no zero termination byte). Each character represents one initialized data byte. Because a string has no implicit termination byte, a single character string in a **DATA.B** directive is the same as an ASCII character constant. For example:

  ```
  DATA.B  "The quick brown fox"
  ```

- A pointer expression consists of the **PTR** keyword followed by an expression. For example:

  ```
  DATA.B  PTR expr
  ```

  The *expr* expression is interpreted as a two-byte constant, and can have values in the range 0..65535. The expression is a general expression, and can be any combination of relocatable symbols, imported or exported symbols, and any applicable operations defined for expressions.

Examples:

The following example defines a look-up table of three constant bytes:

```
SEG     CODE
ORG
APEXP   %lookupTable
```

```
%lookupTable DATA.B  h'12,h'2C,h'39
```

The following example defines a constant string "Hello, World!" and includes a terminating zero byte:

```
             SEG     CODE
             ORG
             APEXP   %hello
%hello       DATA.B  "Hello, World!", 0
```

The following example declares an array of pointers to three functions *funcA*, *funcB*, and *funcC*:

```
             SEG     CODE
             ORG
             IMPORT  funcA, funcB, funcC
             APEXP   %funcTable
%funcTable   DATA.B  PTR funcA
             DATA.B  PTR funcB, PTR funcC
```

ELSE (Conditional Assembly)

The Neuron assembler provides the following directives for conditional control of the assembly of blocks of source lines: **IF**, **IFDEF**, **IFNDEF**, **ELSE**, and **ENDIF**.

A conditional assembly block begins with the **IF**, **IFDEF**, or **IFNDEF** directive, and ends with a matching **ENDIF** directive. A conditional block can contain at most one matching **ELSE** directive.

The **ELSE** directive changes the state of the conditional assembly. If the assembler is currently assembling source lines when it encounters an **ELSE** directive, it skips subsequent source-code lines, up to the matching **ENDIF** directive. If the assembler is currently skipping source-code lines because of an **IF**, **IFDEF**, or **IFNDEF** condition, it begins assembling subsequent lines, up to the matching **ENDIF** directive.

Conditional assembly directives can be nested. Thus, a group of source-code lines between an **IF** directive and a matching **ELSE** or **ENDIF** can contain additional **IF**, **IFDEF**, or **IFNDEF** directives, and **ELSE** directives, as long as there are an identical number of matching **ENDIF** directives. The maximum number of nested directives is five. The total maximum number of conditional **IF**, **IFDEF**, and **IFNDEF** directives for a single source file is 256.

Source-code lines that the assembler skips because of conditional assembly must still conform to the Neuron assembler syntax.

Syntax:

The **ELSE** directive has no arguments and cannot have a label. The **ELSE** directive must be preceded by a matching **IF**, **IFDEF**, or **IFNDEF** directive.

```
ELSE
```

END (Assembly Control)

The **END** directive instructs the Neuron assembler to stop reading the current file and stop compiling. This directive is optional, and is not commonly used.

Do not use the **END** directive if you plan to integrate your assembly code with a Neuron C application using the **#pragma include_assembly_file** Neuron C directive.

Syntax:

The **END** directive has no arguments and does not need a label.

```
END
```

ENDIF (Conditional Assembly)

The Neuron assembler provides the following directives for conditional control of the assembly of blocks of source lines: **IF**, **IFDEF**, **IFNDEF**, **ELSE**, and **ENDIF**.

A conditional assembly block begins with the **IF**, **IFDEF**, or **IFNDEF** directive, and ends with a matching **ENDIF** directive. A conditional block can contain at most one matching **ELSE** directive.

Conditional assembly directives can be nested. Thus, a group of source-code lines between an **IF** directive and a matching **ELSE** or **ENDIF** can contain additional **IF**, **IFDEF**, or **IFNDEF** directives, and **ELSE** directives, as long as there are an identical number of matching **ENDIF** directives. The maximum number of nested directives is five. The total maximum number of conditional **IF**, **IFDEF**, and **IFNDEF** directives for a single source file is 256.

Source-code lines that the assembler skips because of conditional assembly must still conform to the Neuron assembler syntax.

Syntax:

The **ENDIF** directive has no arguments and cannot have a label. The **ENDIF** directive must be preceded by a matching **IF**, **IFDEF**, or **IFNDEF** directive.

```
ENDIF
```

EQU (Equate Symbol)

The **EQU** directive assigns the value of a constant expression to a symbol. The assembler treats subsequent occurrences of the symbol as the value of the constant expression.

Syntax:

The **EQU** directive requires a label and a constant expression argument.

label EQU *const_expr*

Example:

The following example loads TOS with the content of the general-purpose register R0 (base page offset value 8), using the mnemonic name "R0":

```
R0          EQU     8
Example     APEXP           ; ( -- r0)
            push    !R0     ; ( r0 )
            ret             ; return to caller
```

ERROR (Conditional Assembly)

The **ERROR** directive causes the Neuron Assembler to display a user-defined error message. This error leads to an assembly failure, and is counted in the overall assembly statistics.

This directive is useful for managing and validating conditional compilation. This directive is only available with the NodeBuilder FX Development Tool (and later versions).

Syntax:

The **ERROR** directive requires a quoted string as its argument, and cannot have a label.

```
ERROR "string"
```

Example:

The following example stops compilation for the current file and displays a message when neither the *AAA* nor the *BBB* conditional compilation symbols are defined:

```
IFDEF AAA
…
ELSE
IFDEF BBB
…
ELSE
ERROR "You must define either AAA or BBB"
ENDIF
ENDIF
```

EXPORT (Export Symbol)

The **EXPORT** directive is used to export symbols that interface with other Neuron Assembly modules.

An exported symbol must be defined as a label within the assembly file from which it is exported. Exporting the symbol makes it available to the linker. The linker uses exported symbols to resolve references from other object modules. It is possible, and sometimes required, to export and import a symbol within the same assembly file to permit the linker to resolve expressions between separate segments.

See also *APEXP (Application Symbol Export)* on page 134.

Syntax:

The **EXPORT** directive requires either a label or an argument, or both.

```
label EXPORT
      EXPORT symbol
      EXPORT symbol, symbol, ...
label EXPORT symbol
label EXPORT symbol, symbol, ...
```

An argument can be a symbol, or it can be a list of symbols separated by commas. The argument specifies the symbols to be exported. If there is no argument for the directive, it must have a label, and the label becomes the exported symbol. If there are both an argument and a label, the argument specifies the symbols to be exported and the label is a standard label.

Examples:

The following example exports the label *myFunction* as a symbol to another Neuron assembly module:

```
myFunction   EXPORT
```

The following example exports the symbols *Flag1* and *Flag2* to a Neuron assembly module:

```
InFlags      EXPORT  Flag1, Flag2
```

IF (Conditional Assembly)

The Neuron assembler provides the following directives for conditional control of the assembly of blocks of source lines: **IF**, **IFDEF**, **IFNDEF**, **ELSE**, and **ENDIF**.

A conditional assembly block begins with the **IF**, **IFDEF**, or **IFNDEF** directive, and ends with a matching **ENDIF** directive. A conditional block can contain at most one matching **ELSE** directive.

Conditional assembly directives can be nested. Thus, a group of source-code lines between an **IF** directive and a matching **ELSE** or **ENDIF** can contain additional **IF**, **IFDEF**, or **IFNDEF** directives, and **ELSE** directives, as long as there are an identical number of matching **ENDIF** directives. The maximum number of nested directives is five. The total maximum number of conditional **IF**, **IFDEF**, and **IFNDEF** directives for a single source file is 256.

Source-code lines that the assembler skips because of conditional assembly must still conform to the Neuron assembler syntax.

Syntax:

The **IF** directive requires a constant expression as its argument and cannot have a label. The **IF** directive must be followed by a matching **ENDIF** directive.

```
IF const_expr
```

If the expression evaluates to a non-zero value, the Neuron assembler processes the lines following the directive, up to a matching **ELSE** directive, if any. If the expression evaluates to zero, the Neuron assembler skips all of the lines following the directive, up to a matching **ELSE** directive, if any, or to the matching **ENDIF** directive.

IFDEF (Conditional Assembly)

The Neuron assembler provides the following directives for conditional control of the assembly of blocks of source lines: **IF**, **IFDEF**, **IFNDEF**, **ELSE**, and **ENDIF**.

A conditional assembly block begins with the **IF**, **IFDEF**, or **IFNDEF** directive, and ends with a matching **ENDIF** directive. A conditional block can contain at most one matching **ELSE** directive.

Conditional assembly directives can be nested. Thus, a group of source-code lines between an **IF** directive and a matching **ELSE** or **ENDIF** can contain additional **IF**, **IFDEF**, or **IFNDEF** directives, and **ELSE** directives, as long as there are an identical number of matching **ENDIF** directives. The maximum number of nested directives is five. The total maximum number of conditional **IF**, **IFDEF**, and **IFNDEF** directives for a single source file is 256.

Source-code lines that the assembler skips because of conditional assembly must still conform to the Neuron assembler syntax.

Syntax:

The **IFDEF** directive requires a symbol as its argument and cannot have a label. The **IFDEF** directive must be followed by a matching **ENDIF** directive.

```
IFDEF symbol
```

If the symbol is defined, the Neuron assembler processes the lines following the directive, up to a matching **ELSE** directive, if any. If the symbol is not defined, the Neuron assembler skips all of the lines following the directive, up to a matching **ELSE** directive, if any, or to the matching **ENDIF** directive.

You can define symbols anywhere in the assembly source file using the **EQU** directive; see *EQU (Equate Symbol)* on page 140. You can also provide a symbol to the assembler using the **--define** command from the console; see *NAS Command Switches* on page 4.

IFNDEF (Conditional Assembly)

The Neuron assembler provides the following directives for conditional control of the assembly of blocks of source lines: **IF**, **IFDEF**, **IFNDEF**, **ELSE**, and **ENDIF**.

A conditional assembly block begins with the **IF**, **IFDEF**, or **IFNDEF** directive, and ends with a matching **ENDIF** directive. A conditional block can contain at most one matching **ELSE** directive.

Conditional assembly directives can be nested. Thus, a group of source-code lines between an **IF** directive and a matching **ELSE** or **ENDIF** can contain additional **IF**, **IFDEF**, or **IFNDEF** directives, and **ELSE** directives, as long as there are an identical number of matching **ENDIF** directives. The maximum number of nested directives is five. The total maximum number of conditional **IF**, **IFDEF**, and **IFNDEF** directives for a single source file is 256.

Source-code lines that the assembler skips because of conditional assembly must still conform to the Neuron assembler syntax.

Syntax:

The **IFNDEF** directive requires a symbol as its argument and cannot have a label. The **IFNDEF** directive must be followed by a matching **ENDIF** directive.

```
IFNDEF symbol
```

If the symbol is not defined, the Neuron assembler processes the lines following the directive, up to a matching **ELSE** directive, if any. If the symbol is defined, the Neuron assembler skips all of the lines following the directive, up to a matching **ELSE** directive, if any, or to the matching **ENDIF** directive.

You can define symbols anywhere in the assembly source file using the **EQU** directive; see *EQU (Equate Symbol)* on page 140. You can also provide a symbol to the assembler using the **--define** command line on the console; see *NAS Command Switches* on page 4.

IMPORT (Import External Symbol)

The **IMPORT** directive is used to import symbols from other Neuron Assembly modules or a Neuron C application.

An imported symbol must be defined as a label and exported within the module from which it is imported. Exporting a symbol makes it available to the linker. The linker uses imported symbols to resolve references to other object modules. It is possible, and sometimes required, to export and import a symbol within the same assembly file to permit the linker to resolve expressions between code and data segments.

See also *APEXP (Application Symbol Export)* on page 134 and *EXPORT (Export Symbol)* on page 142.

Syntax:

The **IMPORT** directive requires a symbol, or a list of symbols separated by commas. This directive cannot have a label.

```
IMPORT symbol
IMPORT symbol, symbol, ...
```

Each symbol in the argument of the **IMPORT** directive is designated as an imported symbol.

INCLUDE (Assembly Control)

The **INCLUDE** directive is used to include additional assembly source-code modules in the compilation of the current file.

Neuron Assembly include files typically use an **.inc** file extension, and contain commonly used symbol imports and definitions of mnemonics. However, a Neuron Assembly include file can also include code or data.

When the assembler encounters an **INCLUDE** directive, the assembler first attempts to open the file using the file name or path provided. If the name is not an absolute path, then the name is treated as relative to the current directory. If the file is not found, the assembler then attempts to open the file relative to each directory in the search path. To specify a directory name, you can use a double backslash (for example, \\myFolder).

The assembler does not distinguish between absolute paths and relative path. For example, if an **INCLUDE** directive contains the file name "\myFile.inc", the assembler attempts to open the file in the root directory, if it exists. Otherwise, the assembler uses the name as a relative path from each of the directories in the search path in turn until a file named "myFile.inc" is found. If the file is not found, the assembler returns an error and stops compilation. See the *Neuron Tools Errors Guide* for information about Neuron Assembler errors.

Included files can also contain the **INCLUDE** directive. The maximum nesting level of included files is five. The assembler can process a total of 32 input files during one assembly, counting all included files and the original source file.

Syntax:

The **INCLUDE** directive requires a file name enclosed in double-quote characters as its argument. The file name can optionally provide an absolute or relative path to the file.

```
INCLUDE "string"
```

The argument is used as a file name to be included in the assembly process, with the included file appearing in the place of the **INCLUDE** directive. The file name must be specified exactly, including extension. There is no default extension, but the **.inc** extension is recommended.

LIBRARY (Include Library)

The **LIBRARY** directive is used to instruct the linker to use a specific function library when linking an application containing this module. This directive is used when writing assembly code that has a known dependency on a specific function library; the **LIBRARY** directive can be used to express that dependency within the source code itself, without the need to document the dependency or to instruct the development tool about the dependency.

When the Neuron Assembler encounters the **LIBRARY** directive, it accepts the directive and its argument without validation. Errors, such as references to non-existent function libraries, can occur during linking.

Included files can also contain the **LIBRARY** directive. The assembler can process a total of 19 input libraries during one assembly, counting all included files and the original source file.

This directive is only available with the NodeBuilder FX Development Tool (and later versions)

Syntax:

The **LIBRARY** directive requires a library file name enclosed in double-quote characters as its argument. The file name can optionally provide an absolute or relative path to the library.

```
LIBRARY "string"
```

The argument is used as a library name to be passed to the linker. The library name must be specified exactly, including extension. There is no default extension, but the **.LIB** extension is recommended.

The library reference (the double-quoted string argument) can optionally contain pre-defined macros. A macro is a keyword, enclosed in single dollar-signs, which expands to a well-defined value during linking. Using macros allows you to create code (and library references) that is location-independent and can assemble and link on different machines.

Table 61 lists the macros that are supported for the **LIBRARY** directive.

Table 61. System Macros for **LIBRARY** Directive

Macro	Expansion
$LONWORKS$	The local LonWorks directory, generally **c:\LonWorks**. Note that there is no trailing backslash.
IMG	$LONWORKS$\Images
STD	$LONWORKS$\NeuronC\Libraries

Example:

The following example advises the Neuron Linker to include the Echelon provided CENELEC library when linking an application that uses the current module:

```
LIBRARY "$STD$\cenelec.lib"
```

The reference to this library uses the **STD** macro, which resolves to the standard location for Neuron C libraries on the machine that performs the linkage.

LIST (Listing Control)

The **LIST** and **NOLIST** directives control whether assembly source statements are included in a listing file (produced by specifying the **--listing** keyword on the command line; see *Output Files* on page 7). The **LIST** directive instructs the assembler to include all subsequent source statements in the listing file until it encounters a **NOLIST** directive. Unless you specify **NOLIST**, the assembler assumes **LIST** by default. The **LIST** directive itself is shown in the listing file.

See also *NOLIST (Listing Control)* on page 151, *PAGE (Listing Control)* on page 153, and *SUBHEAD (Listing Control)* on page 158.

Syntax:

The **LIST** directive has no arguments and cannot have a label.

```
LIST
```

NOLIST (Listing Control)

The **LIST** and **NOLIST** directives control whether assembly source statements are included in a listing file (produced by specifying the **--listing** keyword on the command line; see *Output Files* on page 7). The **NOLIST** directive instructs the assembler to exclude all subsequent source statements in the listing file, until it encounters a **LIST** directive. Unless you specify **NOLIST**, the assembler assumes **LIST** by default. The **NOLIST** directive itself is shown in the listing file.

See also *LIST (Listing Control)* on page 150, *PAGE (Listing Control)* on page 153, and *SUBHEAD (Listing Control)* on page 158.

Syntax:

The **NOLIST** directive has no arguments and cannot have a label.

```
NOLIST
```

ORG (Segment Control)

The **ORG** directive closes the currently open and active segment, and opens a new segment of the same type. When a segment is closed, it is complete and can be assembled and written to the object output file. However, you do not need to explicitly close a segment because the assembler automatically closes all remaining open segments when the end of the input file is reached, or when the **END** directive is encountered.

Because the Neuron Linker operates on segments as the smallest relocatable unit, it is often best to break a large assembly source file into multiple segments (even of the same type) to allow for more versatile linkage.

See also *SEG (Segment Control)* on page 157.

Syntax:

The **ORG** directive can have one optional argument, but cannot have a label. The argument, if any, can either be a constant expression or the keyword **CONSTRAINED**. The argument specifies the nature of the new segment being opened.

```
ORG
ORG const_expr
ORG CONSTRAINED
```

If no argument is specified, the **ORG** directive opens a new relocatable segment of any size.

The value of the *const_expr* argument is used as the absolute starting address of the segment. The starting address is passed to the linker. The assembler does not check the validity of this starting address. However, the linker does check the validity of the starting address, and checks for address conflicts between multiple absolute segments.

The **CONSTRAINED** keyword instructs the linker to relocate the segment so that it does not cross a Neuron memory page boundary. A Neuron memory page is 256 bytes.

PAGE (Listing Control)

The **PAGE** directive begins a new page in an assembly listing file (produced by specifying the **--listing** keyword on the command line; see *Output Files* on page 7). However, if the current page is empty (not including the heading and subheading, if any), the assembler ignores this directive. The **PAGE** directive itself is not shown in the listing file.

See also *LIST (Listing Control)* on page 150, *NOLIST (Listing Control)* on page 151, and *SUBHEAD (Listing Control)* on page 158.

Syntax:

The **PAGE** directive has no arguments and cannot have a label.

```
PAGE
```

RADIX (Default Radix)

The **RADIX** directive sets the default radix (numeric base) used for constant numeric expressions until another **RADIX** directive is encountered. The default radix is **DECIMAL**.

However, if you include assembly source files into applications that are written in Neuron C using the Neuron C compiler's **#pragma include_assembly_file** directive, the Neuron C compiler always sets the default radix to **HEX**, and an included assembly file cannot change the Neuron C compiler's default radix.

Recommendation: Because the Neuron Assembler does not provide a method to save and restore the current default radix, and because an included file might change the default radix, use the **RADIX** directive only when necessary.

Literal constants without a radix must begin with a numeric digit. For example, if the default radix setting is hexadecimal (**HEX**), you must specify a leading zero if the first non-zero digit of the hexadecimal constant is alphabetic. That is, specify `h'0abc` rather than `h'abc`.

Syntax:

The **RADIX** directive requires an argument for the radix name. This directive cannot have a label.

```
RADIX BINARY
RADIX OCTAL
RADIX DECIMAL
RADIX HEX
```

The radix name can be specified in either lower or upper case.

RES (Reserve Uninitialized Memory)

The **RES** directive reserves a block of memory for uninitialized data.

See also *DATA.B (Reserve Initialized Memory)* on page 135.

Syntax:

The **RES** directive requires a constant expression argument. A label is optional.

```
        RES const_expr
label   RES const_expr
```

The *const_expr* argument designates the number of bytes to reserve as an uninitialized data block.

Although the allocated data is uninitialized, the Neuron Chip's and Smart Transceiver's system firmware always clears all RAM to zero as part of the power-up sequence. Thus, RAM variables do need not to be explicitly initialized to zero.

Example:

The following example reserves space for a two-byte global variable in the RAMFAR segment:

```
                SEG   RAMFAR
                ORG
                APEXP   %globalVar
  %globalVar    RES   d'2
```

RESOURCE (Resource Control)

The **RESOURCE** directive is used by the Neuron C compiler to specify various preferences and requirements to its companion tools (the Neuron Assembler, the Neuron Linker, and the Neuron Exporter). For example, the compiler specifies the number of address table entries, aliases, message tags, and other resources for a device.

Do not specify the **RESOURCE** directive in your Neuron assembly source.

SEG (Segment Control)

The **SEG** directive controls the currently open segment type. Any of the supported segment types (including the one that is currently active) can be selected. The active segment selection can be changed as needed.

The assembler groups assembly instructions and data blocks into one or more segments. A *segment* is a group of assembly instructions and data blocks that are assembled into consecutive bytes of machine instructions and data, at consecutive and ascending addresses. The basic unit for the assembler is a statement, but the basic unit for the linker is a segment. The linker relocates code and data on a segment-by-segment basis.

During assembly:

- One segment of each type is always open.
- Additional code and data can be added only to an open segment.
- Only one of the open segments is the currently active segment.
- The next assembly instruction or data statement encountered in the assembly file is added to the active segment.

The segment type does not affect the assembly of the source lines within it, but does affect the linking of the segment.

The **SEG** directive does not cause additional segments to be opened, but instead selects which of the open segments is active. When the assembler starts, it opens an empty, relocatable segment of each type, and selects the ROM segment as the active segment. Thus, the simplest assembly files need not use either a **SEG** or **ORG** directive if only a single ROM segment is needed.

Typical assembly programs use **SEG CODE** for code and constant data, **SEG RAMFAR** or **RAMNEAR** for RAM variables, and **EEFAR** or **EENEAR** for non-volatile, persistent, variables.

See also *ORG (Segment Control)* on page 152.

Syntax:

The **SEG** directive requires one argument for the type of segment that is being made active. This directive cannot have a label.

```
SEG CODE
SEG EECODE
SEG EEFAR
SEG EENEAR
SEG INITCODE
SEG RAMCODE
SEG RAMFAR
SEG RAMNEAR
SEG ROM
SEG SIDATA
```

Segment type names can be specified in either lower or upper case. See *Segments* on page 25 for information about the segment types.

SUBHEAD (Listing Control)

The **SUBHEAD** directive specifies the text for the subheading line of the assembly listing file (produced by specifying the **--listing** keyword on the command line; see *Output Files* on page 7).

See also *LIST (Listing Control)* on page 150, *NOLIST (Listing Control)* on page 151, and *PAGE (Listing Control)* on page 153.

Syntax:

The **SUBHEAD** directive requires a character or string as its argument and cannot have a label. The characters that follow the **SUBHEAD** directive, and any whitespace that follows the directive, are used as the subheading. The argument for the directive is terminated by the end of the line or by a semicolon.

```
SUBHEAD string
```

If the *string* argument for the **SUBHEAD** directive is empty, the subheading line is cleared. The specified subheading appears on all subsequent pages of the assembly listing until another **SUBHEAD** directive is encountered. The **SUBHEAD** directive itself does not appear in the listing file.

8

System-Provided Functions

This chapter describes the system-provided functions that are used by the Neuron C compiler.

Overview of the Functions

The Neuron firmware provides certain functions for common programming tasks that can make Neuron assembly language programming easier to write and maintain. This chapter describes these system-provided functions.

For each of the system-provided functions, the arguments and return values are described as a series of byte values on the data stack, listed in order with top of stack (TOS) first, following the rules and guidelines described in Chapter 4, *Interfacing with a Neuron C Application*, on page 47, unless explicitly stated otherwise. A function's description explicitly mentions if the function uses the return stack. Unless otherwise specified, all arguments to a function are consumed by the function.

Some functions return Boolean results. For example, the **_equal16** function returns true if two 16-bit operands are equal. For those functions, "true" is reported as one, and "false" is reported as zero.

For each function, the description includes a representation of how the function affects the data stack, its relative location in program memory, and the registers it affects:

- Stack Transformation: Stack affect and stack transition comments follow the recommendations provided in *Documenting Changes to the Stack* on page 40.
- Location: For each function, a location is given as either "near" or "far". For "near" locations, you can call the function with the either the **CALL** or **CALLF** instruction. For "far" locations, you must use the **CALLF** instruction to call the function. A function is designated as "far" if it is "far" on any platform. For a specific platform, if the function address is less than h'2000 (as determined by the generated **.sym** file), then you could use the **CALL** instruction, but you must consider that using this instruction will not allow your code to be portable, nor is it guaranteed to work for future system images on the same platform.
- Registers Affected: Certain functions affect registers, such as P0 or R1. Most functions have no affect on system registers.

_abs16 (Absolute Value, 16 Bit)

This function returns the absolute value of a signed 16-bit integer.

Stack Transformation: (a(2) --|a(2)|)

Location: Near

Registers Affected: None

Example:

This example returns the absolute value of -8.

```
pushd #-d'8          ; (-d'8, -d'1)
call  _abs16         ; (d'8, 0)
```

_abs8 (Absolute Value, 8 Bit)

This function returns the absolute value of a signed integer.

Stack Transformation: (a -- |a|)

Location: Near

Registers Affected: None

Example:

This example returns the absolute value of -8.

```
push #@lb(-d'8)     ; (-d'8)
call _abs8          ; (d'8)
```

_add16 (Add, 16 Bit)

This function adds two 16-bit unsigned integers.

Stack Transformation: (a(2), b(2) -- a(2) + b(2))

Location: Near

Registers Affected: None

Example:

This example adds 50 to 300.

```
pushd #d'300        ; (d'44, 1)
pushd #d'50         ; (d'50, 0, d'44, 1)
call  _add16        ; (d'94, 1)
```

_add16s (Add Signed, 16 Bit)

This function adds two 16-bit signed integers.

Stack Transformation: (a(2), b(2) -- a(2) + b(2))

Location: Near

Registers Affected: None

Example:

This example adds 50 to 300.

```
pushd #d'300        ; (d'44, 1)
pushd #d'50         ; (d'50, 0, d'44, 1)
call  _add16s       ; (d'94, 1)
```

_add_8_16f (Add Fast, 8 Bit to 16 Bit)

This function adds an 8-bit value to a 16-bit value. The “f” stands for “fast” in that if there is no carry, only an 8-bit add is done.

Stack Transformation: (offset, value(2) -- value(2)+offset)

Location: Near

Registers Affected: None

Example:

This example adds 50 to 300.

```
pushd #d'300        ; (d'44, 1)
push  #d'50         ; (d'50, d'44, 1)
call  _add_8_16f    ; (d'94, 1)
```

_adds_8_16 (Add Signed, 8 Bit to 16 Bit)

This function adds an 8-bit signed value to a 16-bit value.

Stack Transformation: (a, b(2) -- a + b(2))

Location: Near

Registers Affected: None

Example:

This example adds -50 to 300.

```
pushd #d'300        ; (d'44, 1)
push  #@lb(-d'50)   ; (-d'50, d'44, 1)
call  _adds_8_16    ; (d'250, 0)
```

_alloc (Allocate Stack Space)

This function allocates space on the data stack by pushing *count* bytes of zeros on the data stack. It can be used to allocate space on the data stack for local variables. For fewer than nine bytes, consider using the **ALLOC** instruction, even though it does not initialize the allocated area to zero.

See also *_dealloc (Deallocate Stack Space and Return)* on page 163.

Stack Transformation: (count -- 0, …, 0)

Location: Near

Registers Affected: None

Example:

This example causes 13 bytes of zeroes to be pushed onto the data stack.

```
push #d'13          ; (13)
call _alloc         ; (0(13))
```

_and16 (And, 16 Bit)

This function returns the bitwise AND of two 16-bit values.

Stack Transformation: (a(2), b(2) -- a(2) & b(2))

Location: Near

Registers Affected: None

Example:

This example performs the operation h'1234 AND h'4321.

```
pushd #h'1234       ; (h'34, h'12)
pushd #h'4321       ; (h,21, h'43, h'34, h'12)
call  _and16        ; (h'20, h'02)
```

_dealloc (Deallocate Stack Space and Return)

This function removes *N* items from the stack and returns to the caller of the caller of this function. Consider using the **DEALLOC** instruction for *N* less than 9. Also, consider using the **_drop_n** function to deallocate local variables and returning to the calling function.

Stack Transformation: (value, a, b, c, … --)

Location: Near

Registers Affected: None

Example:

This example drops 13 bytes from the stack and returns to the caller of *LocalFunction*.

```
        callr LocalFunction
; _dealloc returns here.

LocalFunction
        push #d'13          ; (13, a, b, c, …, m)
        call _dealloc       ; ()
```

_dec16 (Decrement, 16 Bit)

This function decrements a 16-bit value by 1.

Stack Transformation: (value(2) -- value(2)-1)

Location: Near

Registers Affected: None

Example:

This example decrements 300.

```
pushd #d'300        ; (44, 1)
call  _dec16        ; (43, 1)
```

_div16 (Divide, 16 Bit)

This function divides two 16-bit integers to produce a 16-bit integer result. The result of dividing by zero is zero and a software error is logged.

Stack Transformation: (a(2), b(2)-- a(2)/b(2))

Location: Near

Registers Affected: R0, R1, R2, P0, P3

Example:

This example divides 8 by 2.

```
pushd #d'2          ; (d'2, 0)
pushd #d'8          ; (d'8, 0, d'2, 0)
call  _div16        ; (d'4, 0)
```

_div16s (Divide Signed, 16 Bit)

This function divides two signed 16-bit integers to produce a signed 16-bit integer result. The result of dividing by zero is zero and a software error is logged.

Stack Transformation: (a(2), b(2) -- a(2)/b(2))

Location: Near

Registers Affected: R0, R1, R2, P0, P3

Example:

This example divides 8 by 2.

```
pushd #d'2          ; (d'2, 0)
pushd #d'8          ; (d'8, 0, d'2, 0)
call  _div16s       ; (d'4, 0)
```

_div8 (Divide, 8 Bit)

This function divides two integers to produce an integer result. The result of dividing by zero is zero and a software error is logged.

Stack Transformation: (a, b -- a/b)

Location: Near

Registers Affected: R0, R1, R2

Example:

This example divides 9 by 2.

```
pushs #d'2          ; (d'2)
push  #d'8          ; (d'9, d'2)
call  _div8         ; (d'4)
```

_div8s (Divide Signed, 8 Bit)

This function divides two signed integers to produce a signed integer result. The result of dividing by zero is zero and a software error is logged.

Stack Transformation: (a, b -- a/b)

Location: Near

Registers Affected: R0, R1, R2

Example:

This example divides 8 by 2.

```
pushs #d'2          ; (d'2)
push  #d'8          ; (d'8, d'2)
call  _div8s        ; (d'4)
```

_drop_n (Drop N Bytes from Stack)

This function drops *N* bytes from the stack and returns to the calling function.

Stack Transformation: (N, a(N) --)

Location: Near

Registers Affected: None

Example:

This example drops 3 bytes from the stack.

```
pushs #d'3          ; (d'3, a, b, c)
```

```
call  _drop_n       ; ()
```

_drop_n_preserve_1 (Drop N Bytes from Stack and Preserve NEXT)

This function preserves NEXT and removes the next *N* items from the stack.

Stack Transformation: (N, a, b(N) -- a)

Location: Near

Registers Affected: None

Example:

This example drops 3 bytes from the stack and returns *a*.

```
pushs #d'3                ; (3, a, b, c, d)
call  _drop_n_preserve_1  ; (a)
```

_drop_n_preserve_2 (Drop N Bytes from Stack and Preserve NEXT and NEXT+1)

This function preserves two bytes from the stack (in NEXT and the element following NEXT) and removes the next *N* items from the stack.

Stack Transformation: (N, a, b, c(N) – a, b)

Location: Near

Registers Affected: None

Example:

This example drops 3 bytes from the stack and returns *a*, *b*.

```
pushs #d'3                ; (3, a, b, c, d, e)
call  _drop_n_preserve_2  ; (a, b)
```

_drop_n_return_1 (Drop N Bytes from Stack, Preserve NEXT, and Return)

This function preserves NEXT, removes the next *N* items from the stack and returns to the caller of the caller of this function.

Stack Transformation: (N, a, b(N) -- a)

Location: Near

Registers Affected: None

Example:

This example drops 3 bytes from the stack and returns to the caller of *LocalFunction*.

```
            callr LocalFunction         ; (a, b, c, d)
   ; _drop_n_return_1 returns here.     ; (a)

   LocalFunction
            pushs #d'3                  ; (3, a, b, c, d)
            call  _drop_n_return_1      ; (a)
```

_drop_n_return_2 (Drop N Bytes from Stack, Preserve NEXT and NEXT+1, and Return)

This function preserves two bytes on the stack (in NEXT and the element following NEXT) and removes the next *N* items from the stack and returns to the caller of the caller of this function.

Stack Transformation: (N, a, b, c(N) – a, b)

Location: Near

Registers Affected: None

Example:

This example drops 3 bytes from the stack and returns *a*,*b* to *LocalFunction*.

```
            callr LocalFunction         ; (a, b, c, d, e)
   ; _drop_n_return_2 returns here.     ; (a, b)

   LocalFunction
            pushs #d'3                  ; (3, a, b, c, d, e)
            call  _drop_n_return_2      ; (a, b)
```

_equal16 (Equality Test, 16 Bit)

This function returns true if two 16-bit values are equal.

Stack Transformation: (a(2), b(2) -- a(2) == b(2))

Location: Near

Registers Affected: None

Example:

This example determines if h'1234 == h'4321.

```
            pushd #h'1234       ; (h'34, h'12)
            pushd #h'4321       ; (h,21, h'43, h'34, h'12)
```

```
call  _equal16      ; (0)
```

_equal8 (Equality Test, 8 Bit)

This function returns true if two 8-bit values are equal. The native **XOR** instruction can also be used to determine equality between two 8-bit scalars, and can be more efficient than the **_equal8** system function. Note that **XOR** replies with inverse Boolean logic (zero for equality and non-zero for non-equality), while **_equal8** provides positive and simplified logic (1 for equality, 0 for non-equality).

Stack Transformation: (a, b -- a == b)

Location: Near

Registers Affected: None

Example:

This example determines if h'12 == h'21.

```
push #h'12          ; (h'12)
push #h'21          ; (h,21, h'12)
call _equal8        ; (0)
```

_gequ16s (Greater Than or Equal Signed, 16 Bit)

This function returns a >= b, where *a* and *b* are signed 16-bit numbers.

Stack Transformation: (a(2), b(2) -- a(2)>=b(2))

Location: Near

Registers Affected: None

Example:

This example determines if d'10 >= -d'10.

```
pushd #-d'10        ; (-d'10, -d'1)
pushd #d'10         ; (d'10, 0, -d'10, -d'1)
call  _gequ16s      ; (1)
```

_gequ8 (Greater Than or Equal, 8 Bit)

This function returns a >= b.

Stack Transformation: (a, b -- a>=b)

Location: Near

Registers Affected: None

Example:

This example determines if d'10 >= d'20.

```
push #d'20          ; (d'20)
push #d'10          ; (d'10, d'20)
call _gequ8         ; (0)
```

_gequ8s (Greater Than or Equal Signed, 8 Bit)

This function returns a >= b, where *a* and *b* are signed.

Stack Transformation: (a, b -- a>=b)

Location: Near

Registers Affected: None

Example:

This example determines if d'10 >= -d'10.

```
push #@lb(-d'10)    ; (-d'10)
push #d'10          ; (d'10, -d'10)
call _gequ8s        ; (1)
```

_get_sp (Get Stack Pointer)

This function gets the address of an item on the stack. -1 specifies the address of TOS, -2 specifies the address of NEXT, and so on. Note that because TOS is implemented as a dedicated special CPU register, you cannot accessed it through a pointer calculated with **_get_sp**.

Stack Transformation: (offset -- address(2))

Location: Near

Registers Affected: None

Example:

This example gets the address of stack item *c*.

```
push #@lb(-d'3)     ; (-3, a, b, c)
call _get_sp        ; (&c(2), a, b, c)
```

_inc16 (Increment, 16 Bit)

This function increments a 16-bit value by 1.

Stack Transformation: (value(2) -- value(2)+1)

Location: Near

Registers Affected: None

Example:

This example increments 300.

```
pushd #d'300        ; (44, 1)
call  _inc16        ; (45, 1)
```

io_iaccess (Acquire Semaphore)

This function is only available for Series 5000 chips. The function acquires the system synchronization semaphore and returns after the semaphore has been successfully acquired, allowing synchronized access to shared resources from interrupts and application code.

The function takes no arguments and always succeeds, but the time taken until the function returns varies, subject to the state of the semaphore when calling this function.

The Series 5000 chips support one single, binary, semaphore. Multiple calls to this function can lead to deadlock. Use **io_irelease** to release the semaphore.

Stack Transformation: (--)

Location: Far

Registers Affected: None

Example:

This example acquires the semaphore, then modifies a global variable, then releases the semaphore.

```
import   io_iaccess, io_irelease

...                 ; ( data -- )
callf io_iaccess    ; ( data -- )
pop   globalVar     ; ( -- )
callf io_irelease   ; ( -- )
```

io_iaccess_wait (Acquire Semaphore and Wait)

This function is only available for Series 5000 chips. This function is identical to the **io_iaccess()** function, except that it is designed for NodeBuilder debug targets. This function updates the watchdog timer in addition to acquiring the system synchronization semaphore. This function logs a system error if interrupt tasks run in the APP processor.

If an interrupt trigger condition is met while the application is halted at a breakpoint within a lock construct (the Neuron C **__lock{ }** keyword or code bounded by the **io_iaccess()** and **io_irelease** functions), and the interrupt task tries to enter a lock, the Neuron Chip or Smart Transceiver watchdog timer times out, and the device resets. Use the **io_iaccess_wait()** function to avoid the watchdog timer reset. A Neuron C application can use the **#pragma**

deadlock_is_infinite compiler directive to avoid the watchdog timer reset. Do not use this directive or the **io_iaccess_wait()** function for release targets; they are intended only for debug targets.

As with the **io_iaccess()** function, the **io_iaccess_wait()** function takes no arguments and always succeeds, but the time taken until the function returns varies, subject to the state of the semaphore when calling this function.

As with the **io_iaccess()** function, use **io_irelease** to release the semaphore.

Stack Transformation: (--)

Location: Far

Registers Affected: None

Example:

This example acquires the semaphore, then modifies a global variable, then releases the semaphore. This example assumes that interrupts are running on the application processor and that the NodeBuilder debugger will be used. Because you should not use the **io_iaccess_wait()** function in release target code, you should modify all instances of **io_iaccess_wait()** to **io_iaccess()** after debugging is complete.

```
import   io_iaccess, io_access_wait, io_irelease

...                     ; ( data -- )
callf io_iaccess_wait   ; ( data -- )
pop   globalVar         ; ( -- )
callf io_irelease       ; ( -- )
```

io_irelease (Release Semaphore)

This function is only available for Series 5000 chips. The function releases the system synchronization semaphore and returns after the semaphore has been released, completing the sequence of synchronized access to shared resources from interrupts and application code which was started with the **io_iaccess** call that acquired the semaphore.

The function takes no arguments and always succeeds.

The Series 5000 chips support one single, binary, semaphore. Multiple subsequent calls to this function are not supported. Use **io_iaccess** to acquire the semaphore.

Stack Transformation: (--)

Location: Far

Registers Affected: None

Example:

This example acquires the semaphore, then modifies a global variable, then releases the semaphore.

```
import   io_iaccess, io_irelease

...                  ; ( data -- )
callf io_iaccess     ; ( data -- )
pop   globalVar      ; ( -- )
callf io_irelease    ; ( -- )
```

_l_shift16 (Left Shift, 16 Bit)

This function shifts an unsigned 16-bit integer left. *b* must be in the range 0..7.

Stack Transformation: (a(2), b -- a(2)<<b)

Location: Near

Registers Affected: None

Example:

This example shifts 16 left 2.

```
pushs #d'2          ; (d'2)
pushd #d'16         ; (d'16, 0, d'2)
call  _l_shift16    ; (d'64, 0)
```

_l_shift16s (Left Shift Signed, 16 Bit)

This function shifts a signed 16-bit integer left. *b* must be in the range 0..7.

Stack Transformation: (a(2), b -- a(2)<<b)

Location: Near

Registers Affected: None

Example:

This example shifts 16 left 2.

```
pushs #d'2          ; (d'2)
pushd #d'16         ; (d'16, 0, d'2)
call  _l_shift16s   ; (d'64, 0)
```

_l_shift8 (Left Shift, 8 Bit)

This function shifts an unsigned integer left. *b* must be in the range 0..7.

Stack Transformation: (a, b -- a<<b)

Location: Near

Registers Affected: None

Example:

This example shifts 16 left 2.

```
pushs #d'2          ; (d'2)
push  #d'16         ; (d'16, d'2)
call  _l_shift8     ; (d'64)
```

_l_shift8s (Left Shift Signed, 8 Bit)

This function shifts a signed integer left. *b* must be in the range 0..7.

Stack Transformation: (a, b -- a<<b)

Location: Near

Registers Affected: None

Example:

This example shifts 16 left 2.

```
pushs #d'2          ; (d'2)
push  #d'16         ; (d'16, d'2)
call  _l_shift8s    ; (d'64)
```

_l_shift8_<n> (Left Shift by <n>, 8 Bit)

This function shifts an integer left. *n* must be in the range 3..7.

Stack Transformation: (a -- a<<n)

Location: Near

Registers Affected: None

Example:

This example shifts 16 left 3.

```
push #d'16          ; (d'16)
call _l_shift8_3    ; (d'128)
```

_ldP0_fetchl (Load P0 from Fetched Location)

This function takes a 16-bit address, loads two bytes from that location and returns them.

Stack Transformation: (address(2) -- [address(2)+1], [address(2)])

Location: Near

Registers Affected: P0 is modified and contains *address(2)*

Example:

This example loads two bytes from address h'f000, which for this example is assumed to contain h'05 at h'f000 and h'01 at h'f001.

```
pushd #h'f000       ; (h'00, h'f0)
call  _ldP0_fetchl  ; (h'01, h'05)
```

_less16 (Less Than, 16 Bit)

This function returns a < b, where *a* and *b* are unsigned 16-bit numbers.

Stack Transformation: (a(2), b(2) -- a(2)<b(2))

Location: Near

Registers Affected: None

Example:

This example determines if d'10 < d'20.

```
pushd #d'20         ; (d'20, 0)
pushd #d'10         ; (d'10, 0, d'20, 0)
call  _less16       ; (1)
```

_less16s (Less Than Signed, 16 Bit)

This function returns a < b, where *a* and *b* are signed 16-bit numbers.

Stack Transformation: (a(2), b(2) -- a(2)<b(2))

Location: Near

Registers Affected: None

Example:

This example determines if d'10 < -d'10.

```
pushd #-d'10        ; (-d'10, -d'1)
pushd #d'10         ; (d'10, 0, -d'10, -d'1)
call  _less16s      ; (0)
```

_less8 (Less Than, 8 Bit)

This function returns a < b, where *a* and *b* are unsigned.

Stack Transformation: (a, b -- a<b)

Location: Near

Registers Affected: None

Example:

This example determines if d'10 < d'20.

```
push #d'20          ; (d'20)
push #d'10          ; (d'10, d'20)
call _less8         ; (1)
```

_less8s (Less Than Signed, 8 Bit)

This function returns a < b, where *a* and *b* are signed.

Stack Transformation: (a, b -- a<b)

Location: Near

Registers Affected: None

Example:

This example determines if d'10 < -d'10.

```
push #@lb(-d'10)    ; (-d'10)
push #d'10          ; (d'10, -d'10)
call _less8s        ; (0)
```

_log16 (Logical Value, 16 Bit)

This function returns the logical representation of a 16-bit value. If the variable is non-zero, true is returned, otherwise false.

Stack Transformation: (a(2) – a?1:0)

Location: Near

Registers Affected: None

Example:

This example returns the logical value for 5.

```
pushd #d'5          ; (d'5, 0)
call  _log16        ; (1)
```

_log8 (Logical Value, 8 Bit)

This function returns the logical representation of a value. If the variable is non-zero, true is returned, otherwise false.

Stack Transformation: (a – a?1:0)

Location: Near

Registers Affected: None

Example:

This example returns the logical value for 5.

```
pushs #d'5          ; (d'5)
call  _log8         ; (1)
```

_lognot16 (Negated Logical Value, 16 Bit)

This function returns the inverse logical representation of a 16-bit value. If the variable is non-zero, false is returned, otherwise true.

Stack Transformation: (a(2) – a?0:1)

Location: Near

Registers Affected: None

Example:

This example returns the logical inverse of 5.

```
pushd #d'5          ; (d'5, 0)
call  _lognot16     ; (0)
```

_lognot8 (Negated Logical Value, 8 Bit)

This function returns the inverse logical representation of a value. If the variable is non-zero, false is returned, otherwise true.

Stack Transformation: (a – a?0:1)

Location: Near

Registers Affected: None

Example:

This example returns the logical inverse of 5.

```
pushs #d'5          ; (d'5)
call  _lognot8      ; (0)
```

_lshift16_add16 (Left Shift and Add, 16 Bit)

This function takes a 16-bit index, shifts it left *n* times, and adds the result to a base value.

Stack Transformation: (index(2), n, base(2) -- base(2)+(index(2)<<n))

Location: Far

Registers Affected: None

Example:

This example uses a base of h'efe3, an index of 10, and a shift value of 1.

```
pushd #h'efe3           ; (h'e3, h'ef)
pushs #d'1              ; (1, h'e3, h'ef)
pushd #d'10             ; (d'10, 0, 1, h'e3, h'ef)
callf _lshift16_add16   ; (h'f7, h'ef)
```

_lshift8_add16 (Left Shift and Add, Converts 8 Bits to 16 Bits)

This function takes an index, converts it to a 16-bit quantity, shifts it left *n* times, and adds the result to a base value.

Stack Transformation: (index, n, base(2) -- base(2)+(index(2)<<n))

Location: Far

Registers Affected: None

Example:

This example uses a base of h'efe3, an index of 10, and a shift value of 1.

```
pushd #h'efe3           ; (h'e3, h'ef)
pushs #d'1              ; (1, h'e3, h'ef)
push  #d'10             ; (d'10, 1, h'e3, h'ef)
callf _lshift8_add16    ; (h'f7, h'ef)
```

_lshift8by1_add16 (Left Shift By 1 and Add, Converts 8 Bits to 16 Bits)

This function takes an index, converts it to a 16-bit quantity, shifts it left once, and adds the result to a base value.

Stack Transformation: (index, base(2) -- base(2)+(index(2)<<1))

Location: Far

Registers Affected: None

Example:

This example uses a base of h'efe3 and an index of 10.

```
pushd #h'efe3              ; (h'e3, h'ef)
push  #d'10                ; (d'10, h'e3, h'ef)
callf _lshift8by1_add16    ; (h'f7, h'ef)
```

_lshift8by2_add16 (Left Shift By 2 and Add, Converts 8 Bits to 16 Bits)

This function takes an index, converts it to a 16-bit quantity, shifts it left twice, and adds the result to a base value.

Stack Transformation: (index, base(2) -- base(2)+(index(2)<<2))

Location: Far

Registers Affected: None

Example:

This example uses a base of h'efe3 and an index of 5.

```
pushd #h'efe3              ; (h'e3, h'ef)
push  #d'5                 ; (d'5, h'e3, h'ef)
callf _lshift8by2_add16    ; (h'f7, h'ef)
```

_max16 (Maximum Value, 16 Bit)

This function computes the maximum of two unsigned 16-bit integers.

Stack Transformation: (a(2), b(2) -- max(a(2),b(2)))

Location: Near

Registers Affected: None

Example:

This example gets the maximum of 8 and 2.

```
pushd #d'2          ; (d'2, 0)
pushd #d'8          ; (d'8, 0, d'2, 0)
call  _max16        ; (d'8, 0)
```

_max16s (Maximum Signed Value, 16 Bit)

This function computes the maximum of two signed 16-bit integers.

Stack Transformation: (a(2), b(2) -- max(a(2),b(2)))

Location: Near

Registers Affected: None

Example:

This example gets the maximum of -8 and 2.

```
pushd #d'2          ; (d'2, 0)
pushd #@lb(-d'8)    ; (-d'8, -d'1, d'2, 0)
call  _max16s       ; (d'2, 0)
```

_max8 (Maximum Value, 8 Bit)

This function computes the maximum of two unsigned integers.

Stack Transformation: (a, b -- max(a,b))

Location: Near

Registers Affected: None

Example:

This example gets the maximum of 8 and 2.

```
pushs #d'2          ; (d'2)
push  #d'8          ; (d'8, d'2)
call  _max8         ; (d'8)
```

_max8s (Maximum Signed Value, 8 Bit)

This function computes the maximum of two signed integers.

Stack Transformation: (a, b -- max(a,b))

Location: Near

Registers Affected: None

Example:

This example gets the maximum of -8 and 2.

```
pushs #d'2          ; (d'2)
push  #@lb(-d'8)    ; (-d'8, d'2)
call  _max8s        ; (d'2)
```

_memcpy (Copy Memory)

This function performs the equivalent of a *memcpy()* function for a non-zero amount of data. *len* **cannot** be 0. Also, **_memcpy** does not handle overlapping source and destinations.

Stack Transformation: (dst(2), src(2), len --)

Location: Near

Registers Affected: P0 becomes the original "src", and P3 becomes the original "dst"

Example:

This example copies 4 bytes from h'eff0 to h'eff8.

```
pushs #d'4          ; (d'4)
pushd #h'eff0       ; (h'f0, h'ef, d'4)
pushd #h'eff8       ; (h'f8,h'ef,h'f0,h'ef,d'4)
call  _memcpy       ; ()
```

_memcpy1 (Copy Memory from Offset)

This function performs the equivalent of a *memcpy()* function from P0 to P3 for a non-zero amount of data at a given *offset*. *len* **cannot** be 0.

Stack Transformation: (offset, len --)

Location: near

Registers Affected: P0 must be "src" and P3 must be "dst"

Example:

This example copies four bytes at offset 3 from within P0 to P3. That is, P3[3] = P0[3], and so on.

```
pushs #d'4          ; (d'4)
pushs #d'3          ; (d'3, d'4)
call  _memcpy1      ; ()
```

_memset (Set Memory)

This function performs the equivalent of a *memset()* function for a non-zero amount of data. *len* **cannot** be 0.

Stack Transformation: (dst(2), val, len --)

Location: Near

Registers Affected: P0 becomes "dst"

Example:

This example initializes four bytes at h'eff0 to 33.

```
pushs #d'4          ; (d'4)
push  #d'33         ; (d'33, d'4)
pushd #h'eff0       ; (h'f0, h'ef, d'33, d'4)
call  _memset       ; ()
```

_memset1 (Set Memory at P0)

This function performs the equivalent of a *memset()* function for a non-zero amount of data at P0. *len* **cannot** be 0.

Stack Transformation: (val, len --)

Location: Near

Registers Affected: P0 must be "dst"

Example:

This example initializes four bytes at P0 to 33.

```
pushs #d'4          ; (d'4)
push  #d'33         ; (d'33, d'4)
call  _memset1      ; ()
```

_min16 (Minimum Value, 16 Bit)

This function computes the minimum of two unsigned 16-bit integers.

Stack Transformation: (a(2), b(2) -- min(a(2),b(2)))

Location: Near

Registers Affected: None

Example:

This example gets the minimum of 8 and 2.

```
pushd #d'2          ; (d'2, 0)
pushd #d'8          ; (d'8, 0, d'2, 0)
call  _min16        ; (d'2, 0)
```

_min16s (Minimum Signed Value, 16 Bit)

This function computes the minimum of two signed 16-bit integers.

Stack Transformation: (a(2), b(2) -- min(a(2),b(2)))

Location: Near

Registers Affected: None

Example:

This example gets the minimum of -8 and 2.

```
pushd #d'2          ; (d'2, 0)
pushd #@lb(-d'8)    ; (-d'8, -d'1, d'2, 0)
call  _min16s       ; (-d'8, -d'1)
```

_min8 (minimum Value, 8 Bit)

This function computes the minimum of two unsigned integers.

Stack Transformation: (a, b -- min(a,b))

Location: Near

Registers Affected: None

Example:

This example gets the minimum of 8 and 2.

```
pushs #d'2          ; (d'2)
push  #d'8          ; (d'8, d'2)
call  _min8         ; (d'2)
```

_min8s (Minimum Signed Value, 8 Bit)

This function computes the minimum of two signed integers.

Stack Transformation: (a, b -- min(a,b))

Location: Near

Registers Affected: None

Example:

This example gets the minimum of -8 and 2.

```
pushs #d'2          ; (d'2)
push  #@lb(-d'8)    ; (-d'8, d'2)
call  _min8s        ; (-d'8)
```

_minus16s (Negative Signed Value, 16 Bit)

This function negates a signed 16-bit integer.

Stack Transformation: (a(2) -- -a(2))

Location: Near

Registers Affected: None

Example:

This example negates 2.

```
pushd #d'2          ; (d'2, 0)
call  _minus16s     ; (-d'2, -d'1)
```

_mod8 (Modulo, 8 Bit)

This function performs a modulo operation on two unsigned integers to produce an integer result.

Stack Transformation: (a, b -- a%b)

Location: Near

Registers Affected: R0, R1, R2

Example:

This example performs 8 modulo 2.

```
pushs #d'2          ; (d'2)
push  #d'8          ; (d'8, d'2)
call  _mod8         ; (d'0)
```

_mod8s (Modulo Signed, 8 Bit)

This function performs a modulo operation on two signed integers to produce a signed integer result.

Stack Transformation: (a, b -- a%b)

Location: Near

Registers Affected: R0, R1, R2

Example:

This example performs 8 modulo 2.

```
pushs #d'2          ; (d'2)
push  #d'8          ; (d'8, d'2)
call  _mod8s        ; (d'0)
```

_mul16 (Multiply, 16 Bit)

This function multiplies two 16-bit integers to produce a 16-bit result.

Stack Transformation: (a(2), b(2) -- a(2)*b(2))

Location: Near

Registers Affected: R0, P0, P3

Example:

This example multiplies 50 times 40.

```
pushd #d'50          ; (d'50, 0)
pushd #d'40          ; (d'40, 0, d'50, 0)
call  _mul16         ; (d'208, 7)
```

_mul16s (Multiply Signed, 16 Bit)

This function multiplies two 16-bit signed integers to produce a 16-bit result.

Stack Transformation: (a(2), b(2) -- a(2)*b(2))

Location: Near

Registers Affected: R0, P0, P3

Example:

This example multiplies 50 times 40.

```
pushd #d'50          ; (d'50, 0)
pushd #d'40          ; (d'40, 0, d'50, 0)
call  _mul16s        ; (d'208, 7)
```

_mul8 (Multiply, 8 Bit)

This function multiplies two integers to produce an integer result.

Stack Transformation: (a, b -- a*b)

Location: Near

Registers Affected: R0

Example:

This example multiplies 50 times 40.

```
push #d'50           ; (d'50)
push #d'40           ; (d'40, d'50)
call _mul8           ; (d'208, 7)
```

_mul8s (Multiply Signed, 8 Bit)

This function multiplies two signed integers to produce an integer result.

Stack Transformation: (a, b -- a*b)

Location: Near

Registers Affected: R0

Example:

This example multiplies 50 times 40.

```
push #d'50          ; (d'50)
push #d'40          ; (d'40, d'50)
call _mul8s         ; (d'208, 7)
```

_mul_8_16 (Multiply, 8 Bit to 16 Bit)

This function multiplies an 8-bit integer and a signed 16-bit integer to produce a 16-bit result.

Stack Transformation: (a, b(2) -- a*b(2))

Location: Near

Registers Affected: R0, P0, P3

Example:

This example multiplies 50 times 40.

```
pushd #d'50         ; (d'50, 0)
push  #d'40         ; (d'40, d'50, 0)
call  _mul_8_16     ; (d'208, 7)
```

_muls_8_16 (Multiply Signed, 8 Bit to 16 Bit)

This function multiplies a signed 8-bit integer and a signed 16-bit integer to produce a signed 16-bit result.

Stack Transformation: (a, b(2) -- a*b(2))

Location: Near

Registers Affected: R0, P0, P3

Example:

This example multiplies 50 times 40.

```
pushd #d'50         ; (d'50, 0)
```

```
push  #d'40          ; (d'40, d'50, 0)
call  _muls_8_16     ; (d'208, 7)
```

_mul8l (Multiply, 8 Bit with 16 Bit Result)

This function multiplies two unsigned integers to produce a 16-bit result.

Stack Transformation: (a, b -- (a*b)(2))

Location: Near

Registers Affected: P0, P3

Example:

This example multiplies 50 times 40.

```
push #d'50           ; (d'50)
push #d'40           ; (d'40, d'50)
call _mul8l          ; (d'208, 7)
```

_mul8ls (Multiply Signed, 8 Bit with 16 Bit Result)

This function multiplies two signed integers to produce a 16-bit signed result.

Stack Transformation: (a, b -- (a*b)(2))

Location: Near

Registers Affected: P0, P3

Example:

This example multiplies 50 times -40.

```
push #d'50           ; (d'50)
push #@lb(-d'40)     ; (-d'40, d'50)
call _mul8ls         ; (d'48, d'248)
```

_not16 (Not, 16 Bit)

This function returns the one's complement (the bitwise NOT) of a 16-bit integer.

Stack Transformation: (a(2) -- ~a(2))

Location: Near

Registers Affected: None

Example:

This example returns the one's complement of 8.

```
pushd #d'8          ; (d'8, 0)
call  _not16        ; (d'247, d'255)
```

_or16 (Or, 16 Bit)

This function returns the bitwise OR of two 16-bit values.

Stack Transformation: (a(2), b(2) -- a(2) | b(2))

Location: Near

Registers Affected: None

Example:

This example determines h'1234 OR h'4321.

```
pushd #h'1234       ; (h'34, h'12)
pushd #h'4321       ; (h,21, h'43, h'34, h'12)
call  _or16         ; (h'35, h'53)
```

_pop (Pop from TOS and Push to Offset)

This function writes a location on the stack. It can be thought of as an extension to the instruction **POP [DSP][*offset*]**, which writes TOS into the stack at *offset*. For **_pop**, the offset is two less than what would be used in the **POP** instruction.

Stack Transformation: (offset, value -- …, value, …)

Location: Near

Registers Affected: None

Example:

This example does the equivalent of a **POP [DSP][-3]**.

```
                    ; (a, b, c, d, e)
push #@lb(-d'5)     ; (-5, a, b, c, d, e)
call _pop           ; (b, c, d, a)
```

_pop1 (Pop from TOS and Push Short to Offset)

This function is very similar to **_pop**. The difference is that –d'18 is added to the *offset* prior to use. This allows **PUSHS** to be used to push the *offset*, and thus it is more efficient to pass in the *offset*.

Stack Transformation: (offset, value -- …, value, …)

Location: Near

Registers Affected: None

Example:

This example does the equivalent of a **POP [DSP][-9]**.

```
                        ; (a, b, c, d, …, h, i, j, k)
pushs #7                ; (7, a, b, c, d, …, j, k)
call  _pop1             ; (b, c, d, …, h, i, j, a)
```

_popd (Pop from TOS and NEXT, Push to Offset, 16 Bit)

This function is used to write a 16-bit value onto a location on the stack. It is similar to the **_pop** function, but writes two bytes rather than one. The *offset* is two less than what would be used in the **POP** instruction.

Stack Transformation: (offset, value(2) -- …, value(2), …)

Location: Far

Registers Affected: None

Example:

This example does the equivalent of two **POP [DSP][-3]** instructions.

```
                        ; (a, b, c, d, e, f)
push  #@lb(-d'5)        ; (-d'5, a, b, c, d, e, f)
callf _popd             ; (c, d, a, b)
```

_popd1 (Pop from TOS and NEXT, Push Short to Offset, 16 Bit)

This function is the same as **_popd**, except that –d'18 is added to the *offset* before storing. This allows the caller to use a **PUSHS** instruction to load the *offset*.

Stack Transformation: (offset, value(2) -- …, value(2), …)

Location: Far

Registers Affected: None

Example:

This example does the equivalent of two **POP [DSP][-9]** instructions.

```
                        ; (a, b, c, d, …, i, j, k, l)
pushs #7                ; (7, a, b, c, …, i, j, k, l)
callf _popd1            ; (c, d, e, f, …, i, j, a, b)
```

_push (Push from Offset to TOS)

This function is used to fetch a location from the stack. It can be thought of as an extension to the instruction **PUSH [DSP][*offset*]**, which reads a value from the stack and places it in TOS. For **_push**, the offset is two less than what would be used in the **PUSH** instruction.

Stack Transformation: (offset, …, value, … -- value, …, value, …)

Location: Near

Registers Affected: None

Example:

This example does the equivalent of a **PUSH [DSP][-3]**.

```
                        ; (a, b, c, d, e)
push #@lb(-d'5)         ; (-d'5, a, b, c, d, e)
call _push              ; (e, a, b, c, d, e)
```

_push1 (Push Short from Offset to TOS)

This function is very similar to **_push**. The difference is that –d'18 is added to the *offset* prior to use. This allows **PUSHS** to be used to push the *offset*, and thus it is more efficient to pass in the *offset*.

Stack Transformation: (offset, …, value, … -- value, …, value, …)

Location: Near

Registers Affected: None

Example:

This example does the equivalent of a **PUSH [DSP][-9]**.

```
                        ; (a, b, c, d, …, h, i, j, k)
pushs #7                ; (7, a, b, c, d, e)
call  _push1            ; (k, a, b, c, d, …, j, k)
```

_push4 (Copy Top 4 Bytes of Stack, Push to Stack)

This function duplicates the first four bytes on the stack.

Stack Transformation: (a, b, c, d -- a, b, c, d, a, b, c, d)

Location: Near

Registers Affected: None

Example:

```
                              ; (a, b, c, d)
            call _push4       ; (a, b, c, d, a, b, c ,d)
```

_pushd (Push from Offset to TOS and NEXT, 16 Bit)

This function is used to fetch a 16-bit location from the stack. It is similar to **_push** except that it loads two bytes rather than one. For **_pushd**, the *offset* is two less than what would be used in the **POP** instruction.

Stack Transformation: (offset, …, value(2), … -- value(2), …, value(2), …)

Location: Far

Registers Affected: None

Example:

This example does the equivalent of two **PUSH [DSP][-3]** instructions.

```
                              ; (a, b, c, d, e)
            push  #@lb(-d'5)  ; (-d'5, a, b, c, d, e)
            callf _pushd      ; (d, e, a, b, c, d, e)
```

_pushd1 (Push Short from Offset to TOS and NEXT, 16 Bit)

This function is the same as **_pushd**, except that –d'18 is added to the *offset* before fetching. This allows the caller to use a **PUSHS** instruction to load the *offset*.

Stack Transformation: (offset, …, value(2), … -- value(2), …, value(2), …)

Location: Far

Registers Affected: None

Example:

This example does the equivalent of two **PUSH [DSP][-9]** instructions.

```
                              ; (a, b, c, d, …, h, i, j, k)
            pushs #7          ; (7, a, b, c, d, e)
            callf _pushd1     ; (j, k, a, b, c, d, …, j, k)
```

_r_shift16 (Right Shift, 16 Bit)

This function shifts a 16-bit unsigned integer right. *b* must be in the range 0..7.

Stack Transformation: (a(2), b -- a(2)>>b)

Location: Near

Registers Affected: None

Example:

This example shifts 16 right 2.

```
pushs #d'2          ; (d'2)
pushd #d'16         ; (d'16, 0, d'2)
call  _r_shift16    ; (d'4, 0)
```

_r_shift16s (Right Shift Signed, 16 Bit)

This function shifts a signed 16-bit integer right. *b* must be in the range 0..7.

Stack Transformation: (a(2), b -- a(2)>>b)

Location: Near

Registers Affected: None

Example:

This example shifts -16 right 2.

```
pushs #d'2          ; (d'2)
pushd #-d'16        ; (-d'16, -d'1, d'2)
call  _r_shift16s   ; (-d'4, -d'1)
```

_r_shift8 (Right Shift, 8 Bit)

This function shifts an unsigned integer right. *b* must be in the range 0..7.

Stack Transformation: (a, b -- a>>b)

Location: Near

Registers Affected: None

Example:

This example shifts 16 right 2.

```
pushs #d'2          ; (d'2)
push  #d'16         ; (d'16, d'2)
call  _r_shift8     ; (d'4)
```

_r_shift8_<n> (Right Shift <n>, 8 Bit)

This function shifts an unsigned integer right. *n* must be in the range 3..7.

Stack Transformation: (a -- a>>n)

Location: Near

Registers Affected: None

Example:

This example shifts 16 right 3.

```
push #d'16          ; (d'16)
call _r_shift8_3    ; (d'2)
```

_r_shift8s (Right Shift Signed, 8 Bit)

This function shifts a signed integer right. *b* must be in the range 0..7.

Stack Transformation: (a, b -- a>>b)

Location: Near

Registers Affected: None

Example:

This example shifts -16 right 2.

```
pushs #d'2          ; (d'2)
push  #@lb(-d'16)   ; (-d'16, d'2)
call  _r_shift8s    ; (-d'4)
```

_register_call (Call Function from Register)

This function calls a function through a pointer. The supplied address must be one less than the address to be called.

Stack Transformation: (address(2)-1 --)

Location: Near

Registers Affected: None (not including whatever is done by the called function)

Example:

This example invokes a function at address h'f423 (by specifying h'f422).

```
pushd #h'f422           ; (h'f422)
call  _register_call    ; ()
```

_sign_extend16 (Convert 8 Bit to 16 Bit, Preserve Sign)

This function sign extends an 8-bit value. If the most-significant byte (MSB) of the value is 1, then the high byte is 255; otherwise it is 0.

Stack Transformation: (value -- valuelo, valuehi)

Location: Near

Registers Affected: None

Example:

This example sign extends -2.

```
push #@lb(-d'2)          ; (h'fe)
call _sign_extend16      ; (h'fe, h'ff)
```

_sub16 (Subtract, 16 Bit)

This function subtracts two 16-bit integers.

Stack Transformation: (a(2), b(2) -- a(2) - b(2))

Location: Near

Registers Affected: None

Example:

This example subtracts 50 from 300.

```
pushd #d'300         ; (d'44, 1)
pushd #d'50          ; (d'50, 0, d'44, 1)
call  _sub16         ; (d'250, 0)
```

_sub16s (Subtract Signed, 16 Bit)

This function subtracts two signed 16-bit integers.

Stack Transformation: (a(2), b(2) -- a(2) - b(2))

Location: Near

Registers Affected: None

Example:

This example subtracts 50 from 300.

```
pushd #d'300         ; (d'44, 1)
```

```
pushd #d'50          ; (d'50, 0, d'44, 1)
call  _sub16s        ; (d'250, 0)
```

_xor16 (Exclusive OR, 16 Bit)

This function returns the bitwise exclusive OR (XOR) of two 16-bit values.

Stack Transformation: (a(2), b(2) -- a(2) ^ b(2))

Location: Near

Registers Affected: None

Example:

This example determines h'1234 XOR h'4321.

```
pushd #h'1234        ; (h'34, h'12)
pushd #h'4321        ; (h'21, h'43, h'34, h'12)
call  _xor16         ; (h'15, h'51)
```

A

Neuron Assembly Instructions Listed by Mnemonic

This appendix lists the Neuron assembly instructions by mnemonic.

Instructions by Mnemonic

Table 62 lists the Neuron assembly language instructions, ordered by mnemonic. The table also summarizes the operation performed by each instruction. In this operational summary, *D* represents a displacement and *SD* represents a signed displacement.

Table 62. Neuron Assembly Instructions by Mnemonic

Mnemonic	Operand	Operation	Description
ADC		TOS = TOS + [DSP--] + C next element = [++DSP] IF result > 255 C = 1 ELSE C = 0 IP = IP+1	Add NEXT plus CARRY to TOS. Drop NEXT.
ADD		TOS = TOS + [DSP--] IF result > 255 C = 1 ELSE C = 0 IP = IP+1	Add NEXT to TOS. Drop NEXT.
ADD	#literal	TOS = TOS + LITERAL IF result > 255 C = 1 ELSE C = 0 IP = IP+2	Add literal to TOS.
ADD_R		TOS = TOS + [DSP--] IF result > 255 C = 1 ELSE C = 0 Return	Add NEXT to TOS. Drop NEXT. Return to caller.

Mnemonic	Operand	Operation	Description
ALLOC	#literal	[DSP+1] = TOS DSP = DSP + literal IP = IP+1	Move DSP by literal. Literal range is 1 to 8.
AND		TOS = TOS & [DSP--] C = 0 IP = IP+1	Bitwise AND of TOS and NEXT. Drop NEXT.
AND	#literal	TOS = TOS & LITERAL C = 0 IP = IP+2	AND literal with TOS.
AND_R		TOS = TOS & [DSP--] C = 0 Return	Bitwise AND of TOS and NEXT. Drop NEXT. Return to caller.
BR	label	IP = IP+SD IP = IP+2	Branch always. The IP-relative displacement SD ranges from -128 to 127.
BRC	label	IF (CARRY) IP = IP+SD IP = IP+2	Branch on carry. The IP-relative displacement SD ranges from -128 to 127.
BRF	label	IP = absolute address	Branch far.
BRNC	label	IF (!CARRY) IP = IP+SD IP = IP+2	Branch on not carry. The IP-relative displacement SD ranges from -128 to 127.
BRNEQ	#literal, label	IF [TOS <> literal] IP = IP+SD ELSE TOS = [DSP--] IP = IP+3	Branch if TOS not equal to constant. Else drop TOS.
BRNZ	label	IF TOS <> 0 IP = IP+SD TOS = [DSP--] IP = IP+2	Drop TOS, branch if TOS not zero. The IP-relative displacement SD ranges from -128 to 127.

Mnemonic	Operand	Operation	Description
BRZ	label	IF [TOS]=0 IP = IP+SD TOS = [DSP--] IP = IP+2	Drop TOS, branch if TOS was zero. The IP-relative displacement SD ranges from -128 to 127.
CALL	label	[RSP--] = LSB (IP+1) [RSP--] = MSB (IP+1) IP = absolute address	Call function. Absolute address is 13 bits.
CALLF	word-label	[RSP--] = LSB (IP+2) [RSP--] = MSB (IP+2) IP = absolute address	Call far function using a 16-bit address.
CALLR	label	[RSP--] = LSB (IP+1) [RSP--]= MSB (IP+1) IP = IP+SD IP = IP+2	Call a function using relative addressing. The IP-relative displacement SD ranges from -128 to 127.
DBRNZ	label	IF [--[RSP+1]] <> 0 IP = IP +SD ELSE RSP++ IP = IP+2	Decrement the top of the return stack and branch if it is not zero, else drop [RSP]. The IP-relative displacement SD ranges from -128 to 127.
DEALLOC	#literal	DSP = DSP - literal IPH = [++RSP] IPL = [++RSP] IP = IP+1	Move DSP by literal and return. Literal range is 1 to 8.
DEC		TOS = TOS - 1 IF result < 0 C = 1 ELSE C = 0 IP = IP+1	Decrement TOS.

Mnemonic	Operand	Operation	Description
DIV		TEMP = int([DSP] ÷ TOS) TOS = int([DSP] % TOS) [DSP] = TEMP IP++	Divide unsigned integer in NEXT by unsigned integer in TOS. Place unsigned quotient in TOS, unsigned remainder in NEXT. Division by zero results in a quotient of 0xFF and the remainder equal to the dividend. If enabled, it will cause a trap but will not result in a reset.
DROP	NEXT	DSP-- IP = IP+1	Remove NEXT from data stack.
DROP	[RSP]	RSP++ IP = IP+1	Remove top byte from return stack.
DROP	TOS	TOS = [DSP--] IP = IP+1	Remove TOS from data stack.
DROP_R	NEXT	DSP-- Return	Remove NEXT from data stack. Return to caller.
DROP_R	TOS	DSP-- Return	Remove TOS from data stack. Return to caller.
INC		TOS = TOS +1 IF result > 255 C = 1 ELSE C = 0 IP = IP+1	Increment TOS.
INC	[PTR]	[BP + (PTR * 2)]++ IP = IP+1	Increment 2-byte PTR.
MUL		[DSP]:TOS = TOS * [DSP] IP++	Multiply unsigned integer stack elements NEXT:TOS = NEXT * TOS. Place unsigned 16-bit result in TOS and NEXT.

Mnemonic	Operand	Operation	Description
NOP		IP = IP + 1	No operation is done.
NOT		TOS = NOT TOS C = 0 IP = IP+1	Change TOS to its one's complement.
OR		TOS = TOS OR [DSP--] C = 0 IP = IP+1	Bitwise OR of TOS and NEXT. Drop NEXT.
OR	#literal	TOS = TOS | LITERAL C = 0 IP = IP+2	OR literal with TOS.
OR_R		TOS = TOS OR [DSP--] C = 0 Return	Bitwise OR of TOS and NEXT. Drop NEXT. Return to caller.
POP	absolute address	[absolute address] = TOS TOS = [DSP--] IP = IP+3	Pops TOS and stores it at the absolute address.
POP	!D	[BP+D] = TOS TOS = [DSP--] IP = IP+1	Pops TOS and stores it at the BP plus displacement address D. Displacement range is 8..23.
POP	!TOS	[BP+TOS] = [DSP--] IP = IP+1	Pops next element and stores it at the BP plus TOS address. TOS unchanged.
POP	[PTR] [D]	[[BP+(PTR*2)]+D] = TOS TOS = [DSP--] IP = IP+2	Pops TOS and stores it at the address held in 2-byte pointer PTR plus unsigned displacement D of 0 to 255.
POP	[PTR] [TOS]	[[BP+(PTR*2)]+TOS]=[DSP--] IP = IP+1	Pops next element and stores it at the address held in 2-byte pointer PTR plus TOS address.
POP	*CPUREG*	*CPUREG* = TOS TOS = [DSP--] IP = IP+1	Pops TOS and stores it in the specified register. *CPUREG* = FLAGS, RSP, DSP

Mnemonic	Operand	Operation	Description
POP	[DSP][D]	[BP + DSP + D]= TOS TOS = [DSP--] IP = IP+1	Pop DSP relative with displacement D. Negative displacement ranging from -1 to -8.
POPD	[PTR]	[BP + (PTR * 2) + 1]= TOS [BP + (PTR * 2)] = [DSP--] TOS = [DSP--] IP = IP+1	Pop pointer (2 bytes) from data stack into pointer PTR.
POPPUSH		[RSP--] = TOS TOS = [DSP--] IP = IP+1	Pops TOS and pushes it onto top of return stack.
PUSH	absolute address	[++DSP] = TOS TOS = [absolute address] IP = IP+3	Pushes 8-bit value at absolute address onto TOS.
PUSH	#literal	[++DSP] = TOS TOS = literal IP = IP+2	Pushes literal onto data stack.
PUSH	!D	[++DSP] = TOS TOS = [BP+D] IP = IP+1	Pushes 8-bit value at BP plus displacement address D onto TOS. Displacement range D is 8..23.
PUSH	!TOS	[++DSP] = [BP+TOS] IP = IP+1	Pushes 8-bit value at BP plus TOS address onto NEXT. TOS unchanged.
PUSH	[DSP][D]	[DSP+1] = TOS TOS = [BP + DSP + D] DSP++ IP = IP+1	Push DSP relative with displacement D. Negative displacement D ranges from -1 to -8.
PUSH	[RSP]	[++DSP] = TOS TOS = [RSP+1] IP = IP+1	Pushes 8-bit value at top of return stack onto TOS. The return stack is unchanged.

Mnemonic	Operand	Operation	Description
PUSH	[PTR] [D]	[++DSP] = TOS TOS = [[BP+(PTR*2)]+D] IP = IP+2	Pushes 8-bit value at address held in 2-byte pointer PTR plus displacement D (from 0 to 255) onto TOS.
PUSH	[PTR] [TOS]	[++DSP]=[[BP+(PTR*2)]+TOS] IP = IP+1	Pushes 8-bit value at address held in 2-byte pointer PTR plus TOS onto NEXT.
PUSH	*CPUREG*	[++DSP] = TOS TOS = *CPUREG* IP = IP+1	Pushes 8-bit value in specified register onto TOS. *CPUREG* = FLAGS, RSP, DSP, TOS, NEXT

PUSHD	#literal8_1, #literal8_2	[++DSP] = TOS
		[++DSP] = #literal8_1
		TOS = #literal8_2
		IP = IP+3
			Pushes 2 eight-bit literals onto data stack.

PUSHD	#literal16	[++DSP] = TOS [++DSP] = MSB(literal16) TOS = LSB(literal16) IP = IP+3	Pushes one 16-bit constant onto data stack.
PUSHD	[PTR]	[++DSP] = TOS [++DSP] = [BP + (PTR * 2)] TOS = [BP + (PTR * 2) + 1] IP = IP+1	Push content of pointer PTR (2 bytes) onto data stack.
PUSHPOP		[++DSP] = TOS TOS = [++RSP] IP = IP+1	Pops 8-bit value from top of return stack and pushes it onto TOS.
PUSHS	#literal	[++DSP] = TOS TOS = literal IP = IP+1	Pushes literal value from 0 to 7 onto the data stack.
RET		IPH= [++RSP] IPL = [++RSP] IP= IP+1	Return from function.

Mnemonic	Operand	Operation	Description
ROLC		C = TOS MSB TOS = TOS << 1 TOS LSB = OLD C IP = IP+1	Rotate TOS left through CARRY by one.
RORC		C = TOS LSB TOS = TOS >> 1 TOS MSB = OLD C IP = IP+1	Rotate TOS right through CARRY by one.
SBC	NEXT,TOS	TOS = [DSP--] - TOS - C IF result < 0 C = 1 ELSE C = 0 IP = IP+1	Subtract TOS and CARRY from NEXT. Drop NEXT.
SBC	TOS,NEXT	TOS = TOS - [DSP--] - C IF result < 0 C = 1 ELSE C = 0 IP = IP+1	Subtract NEXT and CARRY from TOS. Drop NEXT.
SBR	label	IP = IP+D IP = IP+1	Branch always. Displacement range is 0 to 15.
SBRNZ	label	IF TOS <> 0 IP = IP+D TOS = [DSP--] IP = IP+1	Same as BRNZ, but with displacement range of 0 to 15.
SBRZ	label	IF [TOS]=0 IP = IP+D TOS = [DSP--] IP = IP+1	Same as BRZ, but with displacement range of 0 to 15.

Mnemonic	Operand	Operation	Description
SHL		TOS = TOS <<1 TOS LSB = 0 C = 0 IP = IP+1	Logical shift TOS left by one.
SHLA		C = TOS MSB TOS = TOS << 1 TOS LSB = 0 IP = IP+1	Arithmetic shift TOS left by one.
SHR		TOS = TOS >>1 TOS MSB = 0 C = 0 IP = IP+1	Logical shift TOS right by one.
SHRA		C = TOS LSB TOS = TOS >> 1 TOS MSB = OLD TOS MSB IP = IP+1	Arithmetic shift TOS right by one.
SUB	TOS,NEXT	TOS = TOS - [DSP--] IF result < 0 C = 1 ELSE C = 0 IP = IP+1	Subtract NEXT from TOS. Drop NEXT.
SUB	NEXT,TOS	TOS = [DSP--] - TOS IF result < 0 C = 1 ELSE C = 0 IP = IP+1	Subtract TOS from NEXT. Drop NEXT.
XCH		SWAP (TOS , [DSP]) IP = IP+1	Interchange values of TOS and NEXT.

Mnemonic	Operand	Operation	Description
XOR		TOS = TOS ^ [DSP--] C = 0 IP = IP+1	Bitwise XOR of TOS and NEXT. Drop NEXT.
XOR	#literal	TOS = TOS ^ LITERAL C = 0 IP = IP+2	XOR literal with TOS.
XOR_R		TOS = TOS ^ [DSP--] C = 0 Return	Bitwise XOR of TOS and NEXT. Drop NEXT. Return to caller.

B

Neuron Assembly Instructions Listed by Hexadecimal Opcode

This appendix lists the Neuron assembly instructions by opcode.

Instructions by Opcode

Table 63 lists the Neuron assembly language instructions, ordered by hexadecimal opcode. The table also summarizes the number of operands required for each instruction, the size (in bytes) for each instruction, and the number of processor cycles required for each instruction. Opcodes that are either not in use or are reserved for compiler use are marked with a dash (-).

Table 63. Neuron Assembly Instructions by Opcode

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
00	CALL	1	2	6
01	CALL	1	2	6
02	CALL	1	2	6
03	CALL	1	2	6
04	CALL	1	2	6
05	CALL	1	2	6
06	CALL	1	2	6
07	CALL	1	2	6
08	CALL	1	2	6
09	CALL	1	2	6
0A	CALL	1	2	6
0B	CALL	1	2	6
0C	CALL	1	2	6
0D	CALL	1	2	6
0E	CALL	1	2	6
0F	CALL	1	2	6
10	CALL	1	2	6
11	CALL	1	2	6
12	CALL	1	2	6

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
13	CALL	1	2	6
14	CALL	1	2	6
15	CALL	1	2	6
16	CALL	1	2	6
17	CALL	1	2	6
18	CALL	1	2	6
19	CALL	1	2	6
1A	CALL	1	2	6
1B	CALL	1	2	6
1C	CALL	1	2	6
1D	CALL	1	2	6
1E	CALL	1	2	6
1F	CALL	1	2	6
20	NOP	0	1	1
21	SBR	0	1	1
22	SBR	0	1	1
23	SBR	0	1	1
24	SBR	0	1	1
25	SBR	0	1	1
26	SBR	0	1	1
27	SBR	0	1	1
28	SBR	0	1	1
29	SBR	0	1	1
2A	SBR	0	1	1

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
2B	SBR	0	1	1
2C	SBR	0	1	1
2D	SBR	0	1	1
2E	SBR	0	1	1
2F	SBR	0	1	1
30	SHLA	0	1	2
31	RET	0	1	4
32	BRNC	1	2	2
33	BRC	1	2	2
34	INC [0]	0	1	6
35	INC [1]	0	1	6
36	INC [2]	0	1	6
37	INC [3]	0	1	6
38	RORC	0	1	2
39	ROLC	0	1	2
3A	SHR	0	1	2
3B	SHL	0	1	2
3C	SHRA	0	1	2
3D	NOT	0	1	2
3E	INC	0	1	2
3F	DEC	0	1	2
40	SBRZ	0	1	3
41	SBRZ	0	1	3
42	SBRZ	0	1	3

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
43	SBRZ	0	1	3
44	SBRZ	0	1	3
45	SBRZ	0	1	3
46	SBRZ	0	1	3
47	SBRZ	0	1	3
48	SBRZ	0	1	3
49	SBRZ	0	1	3
4A	SBRZ	0	1	3
4B	SBRZ	0	1	3
4C	SBRZ	0	1	3
4D	SBRZ	0	1	3
4E	SBRZ	0	1	3
4F	SBRZ	0	1	3
50	ADD	0	1	4
51	AND	0	1	4
52	OR	0	1	4
53	XOR	0	1	4
54	ADC	0	1	4
55	SUB TOS,NEXT	0	1	4
56	SBC NEXT,TOS	0	1	4
57	SUB NEXT,TOS	0	1	4
58	ADD #*number*	1	2	3
59	AND #*number*	1	2	3
5A	OR #*number*	1	2	3

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
5B	XOR #*number*	1	2	3
5C	ADD_R	0	1	7
5D	AND_R	0	1	7
5E	OR_R	0	1	7
5F	XOR_R	0	1	7
60	SBRNZ	0	1	3
61	SBRNZ	0	1	3
62	SBRNZ	0	1	3
63	SBRNZ	0	1	3
64	SBRNZ	0	1	3
65	SBRNZ	0	1	3
66	SBRNZ	0	1	3
67	SBRNZ	0	1	3
68	SBRNZ	0	1	3
69	SBRNZ	0	1	3
6A	SBRNZ	0	1	3
6B	SBRNZ	0	1	3
6C	SBRNZ	0	1	3
6D	SBRNZ	0	1	3
6E	SBRNZ	0	1	3
6F	SBRNZ	0	1	3
70	DBRNZ	1	2	5
71	BR	1	2	2
72	BRNZ	1	2	4

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
73	BRZ	1	2	4
74	CALLR	1	2	5
75	BRF	2	3	4
76	BRNEQ	2	3	4 or 6
77	CALLF	2	3	7
78	DEALLOC #8	0	1	6
79	DEALLOC #7	0	1	6
7A	DEALLOC #6	0	1	6
7B	DEALLOC #5	0	1	6
7C	DEALLOC #4	0	1	6
7D	DEALLOC #3	0	1	6
7E	DEALLOC #2	0	1	6
7F	DEALLOC #1 DROP_R TOS	0	1	6
80	PUSHS #0	0	1	4
81	PUSHS #1	0	1	4
82	PUSHS #2	0	1	4
83	PUSHS #3	0	1	4
84	PUSHS #4	0	1	4
85	PUSHS #5	0	1	4
86	PUSHS #6	0	1	4
87	PUSHS #7	0	1	4
88	PUSH !8	0	1	4
89	PUSH !9	0	1	4
8A	PUSH !10	0	1	4

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
8B	PUSH !11	0	1	4
8C	PUSH !12	0	1	4
8D	PUSH !13	0	1	4
8E	PUSH !14	0	1	4
8F	PUSH !15	0	1	4
90	PUSH !16	0	1	4
91	PUSH !17	0	1	4
92	PUSH !18	0	1	4
93	PUSH !19	0	1	4
94	PUSH !20	0	1	4
95	PUSH !21	0	1	4
96	PUSH !22	0	1	4
97	PUSH !23	0	1	4
98	PUSH [0][*offset*]	1	2	7
99	PUSH [1][*offset*]	1	2	7
9A	PUSH [2][*offset*]	1	2	7
9B	PUSH [3][*offset*]	1	2	7
9C	PUSH [0][TOS]	0	1	6
9D	PUSH [1][TOS]	0	1	6
9E	PUSH [2][TOS]	0	1	6
9F	PUSH [3][TOS]	0	1	6
A0	PUSH FLAGS	0	1	4
A1	PUSH [RSP]	0	1	4
A2	PUSH RSP	0	1	4

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
A3	PUSH DSP	0	1	4
A4	PUSH TOS	0	1	3
A5	PUSH NEXT	0	1	4
A6	PUSH !TOS	0	1	4
A7	PUSHPOP	0	1	5
A8	-	-	-	-
A9	-	-	-	-
AA	-	-	-	-
AB	-	-	-	-
AC	-	-	-	-
AD	-	-	-	-
AE	-	-	-	-
AF	-	-	-	-
B0	PUSHD [0]	0	1	6
B1	PUSHD [1]	0	1	6
B2	PUSHD [2]	0	1	6
B3	PUSHD [3]	0	1	6
B4	PUSH #*number*	1	2	4
B5	PUSHD #*expression*	2	3	6
B6	XCH	0	1	4
B7	PUSH *address*	2	3	7
B8	PUSH DSP[-8]	0	1	5
B9	PUSH DSP[-7]	0	1	5
BA	PUSH DSP[-6]	0	1	5

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
BB	PUSH DSP[-5]	0	1	5
BC	PUSH DSP[-4]	0	1	5
BD	PUSH DSP[-3]	0	1	5
BE	PUSH DSP[-2]	0	1	5
BF	PUSH DSP[-1]	0	1	5
C0	ALLOC #1	0	1	3
C1	ALLOC #2	0	1	3
C2	ALLOC #3	0	1	3
C3	ALLOC #4	0	1	3
C4	ALLOC #5	0	1	3
C5	ALLOC #6	0	1	3
C6	ALLOC #7	0	1	3
C7	ALLOC #8	0	1	3
C8	POP !8	0	1	4
C9	POP !9	0	1	4
CA	POP !10	0	1	4
CB	POP !11	0	1	4
CC	POP !12	0	1	4
CD	POP !13	0	1	4
CE	POP !14	0	1	4
CF	POP !15	0	1	4
D0	POP !16	0	1	4
D1	POP !17	0	1	4
D2	POP !18	0	1	4

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
D3	POP !19	0	1	4
D4	POP !20	0	1	4
D5	POP !21	0	1	4
D6	POP !22	0	1	4
D7	POP !23	0	1	4
D8	POP [0][*offset*]	1	2	7
D9	POP [1][*offset*]	1	2	7
DA	POP [2][*offset*]	1	2	7
DB	POP [3][*offset*]	1	2	7
DC	POP [0][TOS]	0	1	6
DD	POP [1][TOS]	0	1	6
DE	POP [2][TOS]	0	1	6
DF	POP [3][TOS]	0	1	6
E0	POP FLAGS	0	1	4
E1	-	-	-	-
E2	POP RSP	0	1	4
E3	POP DSP	0	1	4
E4	DROP TOS	0	1	3
E5	DROP NEXT	0	1	2
E6	POP !TOS	0	1	4
E7	POPPUSH	0	1	5
E8	-	-	-	-
E9	-	-	-	-
EA	-	-	-	-

Hexadecimal OpCode	Mnemonic	Number of Operands	Instruction Size	Number of Cycles
EB	-	-	-	-
EC	MUL	0	1	14
ED	DIV	0	1	14
EE	-	-	-	-
EF	-	-	-	-
F0	POPD [0]	0	1	6
F1	POPD [1]	0	1	6
F2	POPD [2]	0	1	6
F3	POPD [3]	0	1	6
F4	-	-	-	-
F5	DROP_R NEXT	0	1	5
F6	DROP [RSP]	0	1	2
F7	POP *address*	2	3	7
F8	POP [DSP][-8]	0	1	5
F9	POP [DSP][-7]	0	1	5
FA	POP [DSP][-6]	0	1	5
FB	POP [DSP][-5]	0	1	5
FC	POP [DSP][-4]	0	1	5
FD	POP [DSP][-3]	0	1	5
FE	POP [DSP][-2]	0	1	5
FF	POP [DSP][-1]	0	1	5

C

Reserved Keywords

This appendix lists the keywords that are reserved to the Neuron Assembler or the Neuron C compiler.

Keywords

Table 64 lists the reserved keywords for the Neuron Assembler or the Neuron C compiler. Not all of the keywords are available to the programmer. Additional keywords may be reserved for future extension.

Table 64. Reserved Keywords

adc	div	mul	res
add	drop	next	resource
add_r	drop_r	noibits	ret
alloc	dsp	nolist	rolc
altnm	else	nop	rorc
and	end	not	rsp
and_r	endif	offchipmem	rti
apexp	equ	onchipmem	sbc
br	error	or	sbr
brc	export	or_r	sbrnz
brf	flags	org	sbrz
brnc	halt	out	seg
brneq	if	page	shl
brnz	ifdef	pop	shla
brz	ifndef	popd	shr
call	import	poppush	shra
callf	in	ptr	sub
callr	inc	push	subhead
constrained	include	pushd	tos
data.b	ldbp	pushpop	xch
dbrnz	library	pushs	xor
dealloc	list	radix	xor_r
dec			

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

X

www.echelon.com